CNN model Workflow in pytorch

① Dataset Class

Ready-to-use Dataset

* Handling loading & preprocessing data
* is designed to load & preprocess Only One Sample at a time
So we need DataLoader use the dataset class to load multiple Samples (batches, usually 64).

Converts raw data (images, csv) into model ready tensors
* There are 2 ways one is to transform One image, and the other is to transform the whole dataset.

1 image → Tensor with Normalization
all images → Tensor with Normalization

Ready Dataset
DataLoader

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

# Load MNIST dataset without transformations (returns PIL images)
train_dataset = MNIST(root="./data", train=True, download=True)
test_dataset = MNIST(root="./data", train=False, download=True)

# Define DataLoader to handle batch processing
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Check the dataset length
print(f"Train Dataset Size: {len(train_dataset)}")
print(f"Test Dataset Size: {len(test_dataset)}")
```

* Same for CIFAR10, CIFAR 100 :

```python
from torchvision.datasets import CIFAR10
from torch.utils.data import DataLoader

# Load CIFAR-10 dataset without any transformation
train_dataset = CIFAR10(root="./data", train=True, download=True)
test_dataset = CIFAR10(root="./data", train=False, download=True)

# Define DataLoaders for batch processing
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Check dataset sizes
print(f"CIFAR-10 Train Dataset Size: {len(train_dataset)}")
print(f"CIFAR-10 Test Dataset Size: {len(test_dataset)}")
```

* For object Detection Dataset:

Dataset class is different

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

# Define a transform (optional)
transform = transforms.Compose([
    transforms.ToTensor()
])

# Load COCO dataset
train_dataset = CocoDetection(root="./data/train2017",
                              annFile="./data/annotations/instances_train2017.json",
                              transform=transform)

test_dataset = CocoDetection(root="./data/val2017",
                             annFile="./data/annotations/instances_val2017.json",
                             transform=transform)

# Create DataLoader
train_loader = DataLoader(train_dataset, batch_size=4, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=4, shuffle=False)

# Check dataset size
print(f"Train Dataset Size: {len(train_dataset)}")
print(f"Test Dataset Size: {len(test_dataset)}")

# Example: Get an image and its annotations
image, targets = train_dataset[0]
print("Image shape:", image.shape)  # Shape depends on transform
print("Annotations:", targets)  # List of bounding boxes and labels
```

Tensor with Normalization

Often with this Literally can without Normalization

usually at the end of brackets

```python
from torchvision.datasets import CIFAR10
from torchvision import transforms
from torch.utils.data import DataLoader

# Define a transform to convert images to tensors and normalize them
transform = transforms.Compose([
    transforms.ToTensor(),  # Convert to tensor
    transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])  # Normalize
])

# Load CIFAR-10 dataset with transform
train_dataset = CIFAR10(root="./data", train=True, transform=transform,
download=True)
test_dataset = CIFAR10(root="./data", train=False, transform=transform,
download=True)

# Define DataLoaders for batch processing
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Check dataset sizes
print(f"CIFAR-10 Train Dataset Size: {len(train_dataset)}")
print(f"CIFAR-10 Test Dataset Size: {len(test_dataset)}")
```

- ✅ **Use Normalize()** for deep learning models, pretrained models, and better training performance.
  - **Examples:** CIFAR-10, CIFAR-100, ImageNet.
  - **Typical Normalization:**
    - CIFAR-10: mean=[0.4914, 0.4822, 0.4465], std=[0.247, 0.243, 0.261]
    - ImageNet: mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]
  - **Pretrained models (ResNet, VGG, etc.) require it.**
- ❌ **Skip Normalize()** for visualization, simple models, or raw image testing.
  - **Examples:** MNIST, Fashion-MNIST, simple grayscale datasets.
  - **Reason:** These datasets have single-channel grayscale images, where values are already in [0,1] after `ToTensor()`, so additional normalization isn't always needed.

① Each Class in one Folder:

dataset → Folder 1 → .jpg .jpg .jpg ; Folder 2 → .jpg .jpg .jpg

* The main thing is to get ① List of all Paths, ② List of all Labels.

Check the data

① how to read data:
load an image path using (join) with data Path, then read it using (open)

② Finding path of All images
use glob.glob & store it in (image_files)

③ Transform To Tensor step
Load image then apply tensor, then check the shape

④ Check shape of multiple images, do Transformation if necessary on multiple images.

⑤ Dataset class

A PyTorch Dataset class typically consists of **three main functions**:

Function	Purpose
`__init__()`	Loads all image paths & stores class labels.
`__len__()`	Returns the total number of images.
`__getitem__()`	Loads, processes, and returns one image & label.

```python
### ✏ General Structure of a Dataset Class✏
from torch.utils.data import Dataset
from PIL import Image
import glob
class CustomDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir  # Dataset path
        self.transform = transform  # Transformations
        self.class_labels = {"daisy": 0, "dandelion": 1, "rose": 2, "sunflower": 3, "tulip": 4}
        # Get all image paths
        self.image_paths = []
        self.labels = []
        for class_name, label in self.class_labels.items():  # the loop is assign a label to each image
            class_images = glob.glob(f"{root_dir}/{class_name}/*.jpg")  # Find all images
            self.image_paths.extend(class_images)#extend it into the array all images in some point
            self.labels.extend([label] * len(class_images))  # Assign labels, in SEGMENTATION we put here masks paths too not lables

    def __len__(self):
        return len(self.image_paths)  # Total number of images

    def __getitem__(self, idx): #idx is the input or the number of image i want to get with it's label
        image_path = self.image_paths[idx]  # Get image path
        label = self.labels[idx]  # Get label

        # Load image using PIL
        image = Image.open(image_path) # in segmentation we load masks too
        # mask = Image.open(mask_path)

        # Apply transformations (if any)
        if self.transform:
            image = self.transform(image)
        return image, label  # Return processed image & label, in SEGMENTATION we return image, mask
```

Each Class in Folder

⑥ Transform & initialize the dataset

```python
# Define transformations
transform = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize images to 64x64
    transforms.ToTensor(),  # Convert to tensor
])

# Initialize dataset
dataset = CustomDataset(train_data_path, transform=transform)
```

⑦ DataLoader to handle batching automatically:

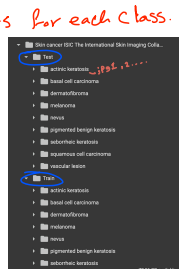
```python
# Define DataLoader
batch_size = 32
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=2)#Your custom dataset

# Get a batch of training images
images, labels = next(iter(dataloader))#next and iter :iterate through the dataloader to iterate over each batch, and next brings to get the next batch
print("Batch shape:", images.shape, "Labels:", labels)
```

```
Batch shape: torch.Size([32, 3, 64, 64]), Labels: tensor([3, 1, 0, 1, 3, 2, 3, 4, 1, 1, 0, 3, 1, 2, 1, 0, 2, 4, 3, 3, 4, 1, 2, 4, 1, 3, 1, 4, 2, 4])
```

✅ Now, instead of reading one image at a time, we process an entire batch! This can be now fed into a model.

② To Folders Train, Test and each of them have Folders for each class.

dataset → Folder 1 → Folder 1.1 → .jpg .jpg ; Folder 1.2 → .jpg .jpg ; Folder 1.3 → .jpg .jpg ; Folder 2 → Folder 2.1, Folder 2.2, Folder 2.3



The data already splitted So the transform step will change, because we can't apply augmentation on test images

⇒ Dataset Class:

```python
from torch.utils.data import Dataset
from PIL import Image
import glob

class SkinCancerDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir  # Dataset path
        self.transform = transform  # Transformations
        self.class_labels = {
            "actinic keratosis": 0, "basal cell carcinoma": 1, "dermatofibroma": 2,
            "melanoma": 3, "nevus": 4, "pigmented benign keratosis": 5,
            "seborrheic keratosis": 6, "squamous cell carcinoma": 7, "vascular lesion": 8
        }
        # Get all image paths
        self.image_paths = []
        self.labels = []
        for class_name, label in self.class_labels.items():
            class_images = glob.glob(f"{root_dir}/{class_name}/*.jpg")  # Find all images
            self.image_paths.extend(class_images)
            self.labels.extend([label] * len(class_images))  # Assign labels

    def __len__(self):
        return len(self.image_paths)  # Total number of images

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]  # Get image path
        label = self.labels[idx]  # Get label

        # Load image using PIL
        image = Image.open(image_path)

        # Apply transformations (if any)
        if self.transform:
            image = self.transform(image)

        return image, label  # Return processed image & label
```

⇒ Transform & path & apply & dataloader:

```python
from torch.utils.data import DataLoader

# Define transformations
transform = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize images
    transforms.RandomRotation(15),  # Rotate images randomly within ±15 degrees , up to 15
    transforms.ColorJitter(brightness=0.2),  # Adjust brightness randomly, up to 0.2
    transforms.ToTensor(),  # Convert to tensor
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5]),  # value for each channel
])
# Validation and testing data typically don't require augmentations, as we only evaluate the model performance on these sets.
# Instead, we apply basic transformations to prepare the images.
transform_valid_test = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize images to 64x64
    transforms.ToTensor(),  # Convert to tensor
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5]),#RGB colors so we normalize each channel of image, if greyscale [0.5] it's 1
])

# Initialize dataset for Train
#if it's not split in folders as train, test folder split it as numbers :
# self.images = os.listdir(img_dir)[:2500] this is in dataset class
#or write it outside for train: train= image_path[:1000] # for test: test = image_path[1000:]
train_path = os.path.join(path, "Skin cancer ISIC The International Skin Imaging Collaboration", "Train")
test_path = os.path.join(path, "Skin cancer ISIC The International Skin Imaging Collaboration", "Test")

train_dataset = SkinCancerDataset(train_path, transform=transform)
test_dataset = SkinCancerDataset(test_path, transform=transform_valid_test)#test data not transformed

# Create DataLoader
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True, num_workers=2)#shuffle in test too

# Get a batch of training images
images, labels = next(iter(train_loader))
print("Batch shape:", images.shape, "Labels:", labels)
```

for test data

Paths apply

Data Loader

we can replace Dataset Class (ImageFolder)

call it

```python
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder # if you have dataset that each class in folder
import os
# Define transformations
transform = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize images
    transforms.RandomRotation(15),  # Rotate images randomly within ±15 degrees
    transforms.ColorJitter(brightness=0.2),  # Adjust brightness randomly
    transforms.ToTensor(),  # Convert to tensor
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5]),  # value for each channel
])

# Validation and testing data typically don't require augmentations, as we only evaluate the model performance on these sets.
# Instead, we apply basic transformations to prepare the images.
transform_valid_test = transforms.Compose([
    transforms.Resize((64, 64)),  # Resize images to 64x64
    transforms.ToTensor(),  # Convert to tensor
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5]),
])

# Initialize dataset for Train
train_path = os.path.join(path, "Skin cancer ISIC The International Skin Imaging Collaboration", "Train")
test_path = os.path.join(path, "Skin cancer ISIC The International Skin Imaging Collaboration", "Test")

train_dataset = ImageFolder(train_path, transform=transform)   #pass ImageFolder here, equivelant to Dataset class that has len and getitem        ## Replac
test_dataset = ImageFolder(test_path, transform=transform_valid_test)                                ## Replaced SkinCancerDataset with ImageFolder

# Create DataLoader
batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True, num_workers=2)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True, num_workers=2)

# Get a batch of training images
images, labels = next(iter(train_loader))
print(f"Batch shape: {images.shape}, Labels: {labels}")
```

(3)

Folders inside folders

```
Smoking_Dataset/
|— Training/ folder
|   |— Training/ folder
|   |   |— notsmoking_0006.jpg
|   |   |— smoking_0007.jpg
|— Validation/
|   |— Validation/
|   |   |— notsmoking_0012.jpg
|   |   |— smoking_0015.jpg
|— Testing/
|   |— Testing/
|   |   |— notsmoking_0032.jpg
|   |   |— smoking_0034.jpg
```

the labels in the image name

(1) Extract labels outside the Class for simplicity

f1
f2
f3
we have labels for each folder
folder 1
folder 2
folder 3

```python
import os
import glob
from tqdm import tqdm

# Define dataset path
dataset_path = os.path.join(path,"Training","Training")
# Get all image paths
train_image_paths = glob.glob(f"{dataset_path}/*.jpg")

# Define dataset path
dataset_path = os.path.join(path, "Validation", "Validation")
# Get all image paths
valid_image_paths = glob.glob(f"{dataset_path}/*.jpg")

# Define dataset path
dataset_path = os.path.join(path,"Testing", "Testing")
# Get all image paths
test_image_paths = glob.glob(f"{dataset_path}/*.jpg")

train_labels = []
valid_labels = []
test_labels = []

for path in tqdm(train_image_paths): #split the images from labels; they are in the same folder i guess
    prefix = path.split("/")[-1].split("_")[0]                  # "/Training/smoking_0007.jpg" - "smoking_0007.jpg" - "smoking"
    label = 1 if prefix == "smoking" else 0                     # Assign label
    train_labels.append(label)                                  # Store label

for path in tqdm(valid_image_paths): #expected because
    prefix = path.split("/")[-1].split("_")[0]
    label = 1 if prefix == "smoking" else 0
    valid_labels.append(label)

for path in tqdm(test_image_paths):
    prefix = path.split("/")[-1].split("_")[0]
    label = 1 if prefix == "smoking" else 0
    test_labels.append(label)
```

(2) Dataset Class:

we already extract labels

```python
from torch.utils.data import Dataset
from PIL import Image

class SmokingDataset(Dataset):
    def __init__(self, image_paths, labels, transform=None):
        self.image_paths = image_paths  # List of image paths
        self.labels = labels  # Corresponding labels
        self.transform = transform  # Transformations to apply

    def __len__(self):
        return len(self.image_paths)  # Total number of images

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]  # Get image path
        label = self.labels[idx]  # Get corresponding label

        # Load image
        image = Image.open(image_path)

        # Apply transformations (if any)
        if self.transform:
            image = self.transform(image)

        return image, label  # Return processed image and its label
```

(3) Transform & initialize data & DataLoader:

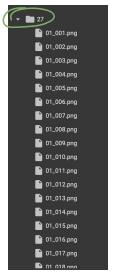
3 folders
3 folders

```python
from torchvision import transforms
from torch.utils.data import DataLoader

# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomRotation(15),
    transforms.ColorJitter(brightness=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5]),
])

transform_valid_test = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.5,0.5,0.5], std=[0.5,0.5,0.5]),
])

# Initialize datasets
train_dataset = SmokingDataset(train_image_paths, train_labels, transform=transform)
valid_dataset = SmokingDataset(valid_image_paths, valid_labels, transform=transform_valid_test)
test_dataset = SmokingDataset(test_image_paths, test_labels, transform=transform_valid_test)

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=32, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Get a batch of training images
images, labels = next(iter(train_loader))
print(f"Batch shape: {images.shape}, Labels: {labels}")
```

(4)

Labels are inside file paths

the data



(1) Extract labels outside the data class

Important

not necessary

```python
# Import necessary libraries
image_dict = {}
# Define the list of file names
from pathlib import Path
from tqdm import tqdm
tqdm.pandas()
import os
import pandas as pd
# Import the necessary module from the Python Imaging Library (PIL).
from PIL import ImageFile
# Enable the option to load truncated images.
# This setting allows the PIL library to attempt loading images even if they are corrupted or incomplete.
ImageFile.LOAD_TRUNCATED_IMAGES = True

MIN_SAMPLES = 100 # only include labels with as many samples
# Initialize empty lists to store file names and labels
file_names = []
labels = []
labels_dict = {'01': 'Akhal-Teke', '02': 'Appaloosa', '03': 'Orlov Trotter', '04': 'Vladimir Heavy Draft', '05': 'Percheron',
'06': 'Arabian', '07': 'Friesian'}
# Iterate through all image files in the specified directory
#for file in sorted(Path('/kaggle/input/horse-breeds/').glob('*.png')):
for file in sorted(Path('/content/data/datasets/olgabelitskaya/horse-breeds/versions/2').glob('*.png')):
    # Check number of such files in a directory
    sample_dir = '/'.join(str(file).split('/')[:-1])+'/'
    num_files_in_dir = len([1 for _, _, x in os.walk(sample_dir)][0])
    if num_files_in_dir >= MIN_SAMPLES:
        file_names.append(str(file))  # Add the file path to the list
        label = labels_dict[str(file).split('/')[-1][:2]]  # Extract the label from the file
        labels.append(label)  # Add the label to the list
print(len(file_names), len(labels))

# Create a pandas dataframe from the collected file names and labels
df = pd.DataFrame.from_dict({"label": labels, "image": file_names})

labels = list(df['label'].values)
print(df.shape)
```

(2) Dataset class & Transform & Initialize

```python
from torch.utils.data import Dataset
from PIL import Image

class HorseDataset(Dataset):
    def __init__(self, image_paths, labels, transform=None):
        self.image_paths = image_paths  # List of image paths
        self.labels = labels  # Corresponding labels
        self.transform = transform  # Transformations to apply

    def __len__(self):
        return len(self.image_paths)  # Total number of images

    def __getitem__(self, idx):
        image_path = self.image_paths[idx]  # Get image path
        label = self.labels[idx]  # Get corresponding label

        # Load image
        image = Image.open(image_path)

        # Apply transformations (if any)
        if self.transform:
            image = self.transform(image)

        return image, label  # Return processed image and its label
```

```python
# Define transformations
transform = transforms.Compose([
    transforms.Grayscale(),  # Grayscale
    transforms.Resize((64, 64)),  # Resize all images to 64x64
    transforms.ToTensor(),  # Convert image to tensor
])

# Initialize th
dataset = HorseDataset(image_paths=file_names, labels=labels, transform=transform)
```

(3) Data loader:

```
# Create DataLoaders
data_loader = DataLoader(dataset, batch_size=32, shuffle=True)

# Get a batch of training images
images, labels = next(iter(data_loader))
print("Batch shape:", images.shape, "Labels:", labels)
```

```
Batch shape: torch.Size([32, 1, 64, 64]), Labels: ['Appaloosa', 'Akhal-Teke', 'Appaloosa', 'Akhal-Teke', 'Friesian', 'Friesian', 'Akhal-Teke', 'Orlov Trotter',

print(labels[:8])

['Appaloosa', 'Akhal-Teke', 'Appaloosa', 'Akhal-Teke', 'Friesian']
```

② Model Class

Class CustomModule(nn.Module):

Define layers (__init__ method):

def __init__(self):

① The model

super(CustomModel, self).__init__()

② Conv layer + Activation + Pooling

* Can be repeated many times: self.Conv1, self.Conv2.....
self.relu1, self.relu2.....
self.pool1, self.pool2.....

⇒ Conv layer consist of (in_channels = 1, out_channels = 16, kernel_size = 3, stride = 1, padding = 1)

- in_channels: 1 for greyscale datasets, 3 for RGB datasets,
- out_channels: usually you choose it 16 or 32
- kernel_size: it's the filter size usually 3X3.
- Stride: You can choose it but better to be 1.
- padding: if it set to 1 that means the same padding (weight & height remains the same)
if it not written it will affect the size by subtract it by 2 ⇒ 32 - 2 = 30 width
30 height

⇒ Activation function

- use ReLu to add non-linearity (always ReLu).

⇒ Pooling

- reduce spatial dimensions (height, width) the output images
- You can check it by the formula, or trial-and-error ⇒

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1$$

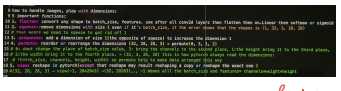
n_{in}: number of input features
n_{out}: number of output features
k: convolution kernel size
p: convolution padding size
s: convolution stride size

- Either you choose MaxPool2D or AvgPool2D
- Essentially consist of Kernal_size, & Stride is optional.
decrease it — increase it

③ Fully connected layer

* Consist of dimension changing, Linear layer & activation function.

⇒ change the dimensions is important to prepare images for Linear.



Manipulating tensor shapes is essential in deep learning. PyTorch provides several functions to modify tensor dimensions.

1 Flatten

- Converts **any shape** to (batch_size, features).
- **Example:**
(batch_size, channels, height, width) → (batch_size, features)

2 Squeeze

- **Removes dimensions** with size 1.
- **Example:**
(1, 32, 3, 28, 28) → (32, 3, 28, 28)

3 Unsqueeze

- **Adds a dimension** with size 1 at a specified position.
- **Example:**
(3, 28, 28) → (1, 3, 28, 28)

4 Permute

- **Reorders the dimensions** of a tensor by specifying the **new order of indices**.
- **Example:**
(32, 28, 28, 3) → permute(0, 3, 1, 2) → (32, 3, 28, 28)

5 View (works similar to reshape)

- **Reshapes a tensor freely** while maintaining the same number of elements.
- **Example:**
(32, 28, 28, 3) → view(-1, 28*28*3) → (32, 28283)

Operation	Function	Purpose	Example Transformation
Flatten	.flatten()	Convert tensor to (batch, features)	(32, 3, 28, 28) → (32, 3*28*28)
Squeeze	.squeeze()	Remove dims of size 1	(1, 3, 28, 28) → (3, 28, 28)
Unsqueeze	.unsqueeze(dim)	Add a dim of size 1	(3, 28, 28) → (1, 3, 28, 28)
Permute	.permute(dims)	Change order of dimensions	(32, 28, 28, 3) → (32, 3, 28, 28)
View	.view(shape)	Reshape freely	(32, 28, 28, 3) → (32, 28*28*3)

⇒ Linear layer is typically used as the final layer in CNNs for Classification. (nn.Linear)

- Consist of (channels * height * width, classes)
if you don't know the dimension use trial-and-error, or the formula.
⇒ MNist classes 0→9 (10)
⇒ CIFAR10 classes (10)
⇒ CIFAR100 classes (100)

might be different based on the Goal

⇒ Activation functions

- Use Softmax for multi-class classification, converts logits into a probability.
⇒ Typically paired with CrossEntropyLoss (which often softmax applied it internally)
- use Sigmoid for Binary-Class classification, or multi-label classification (image have multi tags).
⇒ Commonly used with Binary Cross Entropy loss
- Use Tanh for Regression, it's range (-1, 1)
⇒ non-probabilistic output.

⇒ Full:

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

class CustomModel(nn.Module):
    def __init__(self):
        """
        Define all layers in the model.
        """
        super(CustomModel, self).__init__()

        # Convolutional Layer + Activation + Pooling
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1) #out_channnels= usually you choose it 16 or 32 so 28*28
        #TODO:   #You remembered something?
        self.relu = nn.ReLU()

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Fully Connected Layer
        #What is the expected size after doing the maxpooling? Check above to know the original size
        self.fc = nn.Linear(16 * 14 * 14, 10)  # Output 10 classes,#14*14 using the formula

        # ???? Layer
        #TODO:  #Which layer is this , multiclass so we use Softmax
        self.softmax = nn.Softmax(dim=1)
```

```python
def forward(self, x): #now we use them not only determine them
    """
    Define the forward pass (how data flows through the model).
    """
    x = self.conv1(x)  # Convolution
    x = self.relu(x)#TODO  # Activation function?
    x = self.pool(x)  # Pooling
    x = x.view(-1, 16*14*14 )#TODO#What should we do before using fully connected layer? -1 for whatevere input batching,
    # 16*14*14 16 is the number of out_channels and 14,14 from the formula
    x = self.fc(x)  # Fully connected layer
    x = self.softmax(x)#TODO  # Convert logits to probabilities
    return x  # Returns probability distribution
```

nn.Linear (16*14*14, 10)

And we can use pretrained model (AlexNet, ResNet, UNet.....)

③ Training Loop

① Create a function consist of model, dataloader, criterion, optimizer, device:

⇒ set the model to training mode & set loss, correct, total to Zero

⇒ Begin the Loop for images, labels in tqdm(dataloader):

- Transfer images, labels to the GPU
- Forward pass & compute loss.
- update weights (Backpropagation)
- Calculate Total Loss for accuracy
- Track accuracy (use softmax here if you use CrossEntropyLoss)

⇒ Compute accuracy outside the for loop but under the function

```python
from tqdm import tqdm    # Shows progress bar

# 🔷 Training Loop
def train_one_epoch(model, dataloader, criterion, optimizer, device):
    model.train()  # Set model to training mode
    total_loss = 0
    correct = 0
    total = 0

    for images, labels in tqdm(dataloader):
        images, labels = images.to(device), labels.to(device)


        outputs = model(images)  # Forward pass
        loss = criterion(outputs, labels)  # Compute loss

        optimizer.zero_grad()  # Reset gradients
        loss.backward()  # Backpropagation
        optimizer.step()  # Update weights

        total_loss += loss.item()

        # Track accuracy
        outputs = torch.softmax(outputs, dim=1)
        predictions = outputs.argmax(dim=1)  # Get class with highest probability
        correct += (predictions == labels).sum().item()
        total += labels.size(0)

    avg_loss = total_loss / len(dataloader)
    accuracy = 100 * correct / total  # Compute accuracy in percentage
    return avg_loss, accuracy
```

→ In the Running we have to specify which criterion, optimizer we have to use for each goal.

multi-class classification ⇒ CrossEntropyLoss()

Binary-class // +Sigmoid → BCELoss()

// without Sigmoid → BCEWithLogitsLoss()

Regression or AutoEncoder ⇒ MSELoss()

③ Validation Loop

★ Same as Training Loop, except updating weights ★

① Create a function consist of model, dataloader, criterion, optimizer, device:

⇒ Set the model to Validation mode & Set Loss, correct, total to Zero

⇒ Disable gradient computation (with torch.no_grad()) above the loop.

⇒ Begin the Loop for images, labels in (dataloader): → without tqdm

- Transfer images, labels to the GPU ★★
- Forward pass & Compute loss.
- Calculate Total Loss for accuracy
- Track accuracy (use Softmax here if you use CrossEntropyLoss)

⇒ Compute accuracy outside the for loop but under the function

```python
# 🔷 Validation Loop
def validate(model, dataloader, criterion, device):
    model.eval()  # Set model to evaluation mode
    total_loss = 0
    correct = 0
    total = 0

    with torch.no_grad():  # Disable gradient computation
        for images, labels in dataloader:
            images, labels = images.to(device), labels.to(device)

            outputs = model(images)  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            total_loss += loss.item()

            # Compute accuracy
            outputs = torch.softmax(outputs, dim=1)
            predictions = outputs.argmax(dim=1)  # Get predicted class
            correct += (predictions == labels).sum().item()
            total += labels.size(0)

    avg_loss = total_loss / len(dataloader)
    accuracy = 100 * correct / total  # Compute accuracy in percentage
    return avg_loss, accuracy
```

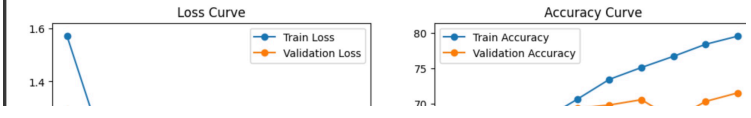
Running Training + store & plot metrices

1. Initialize the model as object not a class, then Transfer it to GPU.
2. Define Loss function & optimizer & # of epochs
3. Training process : Loop for epoch
4. Lists to store metrices, then store them & print
5. plot Loss curve, plot accuracy curve
6. You can show predicted labels on images as function

```python
import torch.optim as optim

# Initialize the model
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = CNNModel().to(device)

# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()  # Multi-class Classification loss (Input: Logits, not probabilities)
optimizer = optim.Adam(model.parameters(), lr=0.001)  # Adam optimizer
num_epochs = 10 # Number of epochs


# Lists to store metrics
train_losses = []
val_losses = []
train_accuracies = []
val_accuracies = []

# Training process
for epoch in range(num_epochs):
    train_loss, train_accuracy = train_one_epoch(model, train_loader, criterion, optimizer, device)
    val_loss, val_accuracy = validate(model, test_loader, criterion, device)

    # Store metrics
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    train_accuracies.append(train_accuracy)
    val_accuracies.append(val_accuracy)

    print(f"Epoch {epoch+1}/{num_epochs}: "
          f"Train Loss={train_loss:.4f}, Train Accuracy={train_accuracy:.2f}%, "
          f"Val Loss={val_loss:.4f}, Val Accuracy={val_accuracy:.2f}%")
```

5

```python
import matplotlib.pyplot as plt

# Plot loss curve
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs+1), train_losses, label="Train Loss", marker='o')
plt.plot(range(1, num_epochs+1), val_losses, label="Validation Loss", marker='o')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss Curve")
plt.legend()

# Plot accuracy curve
plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs+1), train_accuracies, label="Train Accuracy", marker='o')
plt.plot(range(1, num_epochs+1), val_accuracies, label="Validation Accuracy", marker='o')
plt.xlabel("Epochs")
plt.ylabel("Accuracy (%)")
plt.title("Accuracy Curve")
plt.legend()

plt.show()
```



6

```python
import matplotlib.pyplot as plt
import numpy as np

# Function to display images with predicted labels
def show_predictions(model, dataloader, device, num_images=10):
    model.eval()  # Set to evaluation mode
    images, labels = next(iter(dataloader))  # Get a batch
    images, labels = images.to(device), labels.to(device)

    with torch.no_grad():  # Disable gradient computation
        outputs = model(images)
        predictions = outputs.argmax(dim=1)  # Get predicted class

    # CIFAR-10 class names
    classes = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer',
               'Dog', 'Frog', 'Horse', 'Ship', 'Truck']

    # Plot images with predictions
    fig, axes = plt.subplots(2, 5, figsize=(10, 5))
    for i, ax in enumerate(axes.flat[:num_images]):
        img = images[i]
        img = np.transpose(img.cpu().numpy(), (1, 2, 0))  # Convert to (H, W, C)

        ax.imshow(img)
        ax.set_title(f"Pred: {classes[predictions[i].item()]}\nTrue: {classes[labels[i].item()]}")
        ax.axis("off")

    plt.show()

# Show predictions
show_predictions(model, test_loader, device)
```

The labs included:

1. Pytorch Basics
2. Image classification
   - from scratch (CustomModel, CNNModel)
   - pretrained CNN (AlexNet)

Copy - day1-2 ⇒ (2)
** New Right - day1-2 (AlexNet)

3. Image Generation using AutoEncoder
   image → image
   - ⇒ encoder → compressed images (dimensionality reduction)
   - ⇒ decoder → image generator (numbers → image)
4. Custom Dataset class & Image Augmentation

* Batch Normalization is necessary when increase or decrease the size of images.

5. object detection
6. Image Segmintation

Image Search

⇒ might be:
- image corralization
- image Reconstruction as example

Greyscale image (1 channel) → RGB image (might use Segmentation)