

## PoC – PyTorch

### 1. Introduction

PyTorch is an open-source deep learning framework developed by Facebook AI Research known for its flexibility, excellent support for GPUs and compatibility with the popular Python high-level programming language favored by machine learning developers and data scientists.

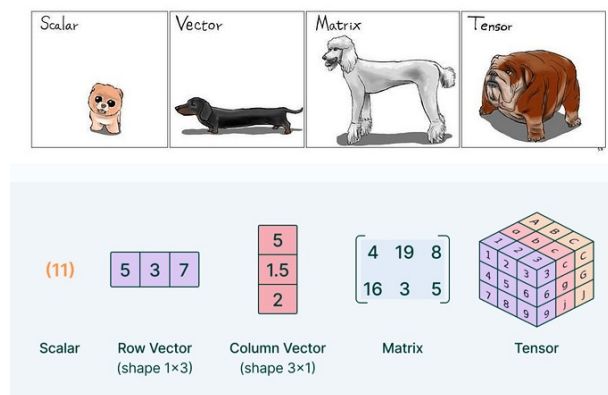
The framework combines the efficient and flexible GPU-accelerated back-end libraries from Torch with an intuitive Python front-end that focuses on rapid prototyping, readable code, and support for the widest possible variety of deep learning models.

### 2. Principles

#### 2.1 Tensors

Tensors are a core PyTorch datatype, similar to a multidimensional array, used to store and manipulate the inputs and outputs of a model, as well as the model's parameters. Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs to accelerate computing.

In short, scalars are single numbers and are an example of a 0th-order tensor, vectors are ordered arrays of single numbers and are an example of 1st-order tensor, matrices are rectangular arrays consisting of numbers and are an example of 2nd-order tensors (Figure 1).



*Figure 1: Scalar, Vector, Matrix and Tensor Illustration*

#### 2.2 Graphs

Neural networks transform input data by applying a collection of nested functions to input parameters.

Graphs are data structures consisting of connected nodes (called vertices) and edges. Every modern framework for deep learning is based on the concept of graphs, where Neural Networks are represented as a graph structure of computations.

### 3. PyTorch Use Cases

PyTorch has many applications in fields such as natural language processing (for the development of language translator, language modeling), in computer vision (image classification, semantic segmentation, object detection) and in machine learning and reinforcement learning.

The main application we will be focusing on is image classification based on the example below.

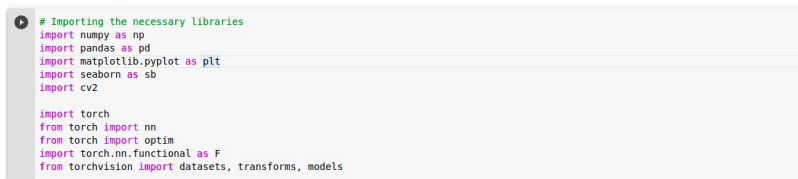
The example code contains an image classification of 20 species of flowers.

We will be analyzing and discussing the use of PyTorch and how it enables us to leverage its functions in order to develop the learning algorithm and to classify the input data.

The dataset can be found here (<https://www.robots.ox.ac.uk/~vgg/data/flowers/102/index.html>)

#### 3.1 Importing PyTorch functionalities

The main functionalities we will focus on will be the ones made available by the torch library as seen in Figure 1.



```
# Importing the necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sb
import cv2

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

Figure 2: Importing the torch library

- *import torch*: the package contains data structures for multi-dimensional tensors and defines mathematical operations over these tensors
- *from torch import nn*: nn refers to neural network and it will allow us to use and benefit from multiple functions on built-in, established neural network architectures providing us access to all the layers of the networks in cases where changes need to be integrated
- *from torch import optim*: the optimize package implements various optimization algorithms
- *import torch.nn.functional as F*: it will help us integrate convolution functions into the neural network architecture
- *from torchvision import datasets, transforms, models*: The *torchvision* package consists of popular datasets, model architectures, and common image transformations for computer vision;
  - *transforms* will help us perform image augmentation before feeding the samples into the learning model;
  - *models* contains definition of models for addressing different tasks including image classification and it offers pre-trained weights which can boost the learning capabilities of the model;
  - *datasets* module includes many built-in datasets, as well as utility classes for building your own datasets. The dataset we will be using in the following demonstration is part of a larger dataset provided by PyTorch.

## 3.2 The Dataset

The entire dataset is separated into training, validation and testing as seen in Figure 3. We will feed the training set into the learning model in order for it to learn and generalize the characteristic features of each class of flowers. The validation set consists of samples that have been held back in order to be used to give an unbiased estimate of how well the model has learned while tuning its hyper-parameters using the optimization methods made available by the optimize package imported in the beginning. Finally, the test set is used to provide an unbiased evaluation of the final model and its ability to classify the input data.

```
[ ] data_dir = 'flowers'
    train_dir = data_dir + '/train'
    valid_dir = data_dir + '/valid'
    test_dir = data_dir + '/test'
```

Figure 3: Train, validation and test sets

## 3.3 Image Augmentation

Image augmentation techniques are commonly used in machine learning in order to increase the size of the dataset, especially for training. This step will help increase the accuracy of the model by enhancing its ability to recognize new variations of the data. The techniques include small adjustments to the already existing images such as cropping, resizing, changing brightness levels, changing the orientation.

`torchvision.transforms` module makes available a series of image transformations which can be chained together using *Compose* (Figure 4).

```
# Define transforms for the training, validation, and testing sets
training_transforms = transforms.Compose([transforms.RandomRotation(30),
                                         transforms.RandomResizedCrop(224),
                                         transforms.RandomHorizontalFlip(),
                                         transforms.ToTensor(),
                                         transforms.Normalize([0.485, 0.456, 0.406],
                                                             [0.229, 0.224, 0.225])])

validation_transforms = transforms.Compose([transforms.Resize(256),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize([0.485, 0.456, 0.406],
                                                             [0.229, 0.224, 0.225])])

testing_transforms = transforms.Compose([transforms.Resize(256),
                                        transforms.CenterCrop(224),
                                        transforms.ToTensor(),
                                        transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

# Loading the datasets with ImageFolder
training_dataset = datasets.ImageFolder(train_dir, transform=training_transforms)
validation_dataset = datasets.ImageFolder(valid_dir, transform=validation_transforms)
testing_dataset = datasets.ImageFolder(test_dir, transform=testing_transforms)


# Using the image datasets and the transforms, define the dataloaders
train_loader = torch.utils.data.DataLoader(training_dataset, batch_size=64, shuffle=True)
validate_loader = torch.utils.data.DataLoader(validation_dataset, batch_size=32)
test_loader = torch.utils.data.DataLoader(testing_dataset, batch_size=32)
```

Figure 4: Image augmentation techniques using `torchvision.transforms`

- `transform.RandomRotation(30)`: rotates the image by the given 30 degrees angle
- `transform.RandomResizedCrop(224)`: Crop a random portion of image and resize it to a square crop based on the given number as side length
- `transform.RandomHorizontalFlip()`: randomly flip the input horizontally
- `transforms.toTensor()`: Converts a PIL Image or ndarray to tensor and scale the values accordingly
- `transform.Normalize()`: Normalizes a tensor image with mean and standard deviation (the provided values are the default ones provided in the Torch documentation)
- `transforms.CenterCrop(224)`: crops the given image at the center and forms a new image with the sides equal to the given number

- `transforms.Resize(256)`: resizing the given image at the size provided in the parenthesis

The association between the name of each of the classes and their respective number is done via a `flowers_to_name.json` file that holds a dictionary data structure containing the number and the species of each class of flower (Figure 5).



```
1 {  
2   "1": "pink primrose",  
3   "2": "hard-leaved pocket orchid",  
4   "3": "canterbury bells",  
5   "4": "sweet pea",  
6   "5": "english marigold",  
7   "6": "tiger lily",  
8   "7": "moon orchid",  
9   "8": "bird of paradise",  
10  "9": "monkshood",  
11  "10": "globe thistle",  
12  "11": "snapdragon",  
13  "12": "colt's foot",  
14  "13": "king protea",  
15  "14": "spear thistle",  
16  "15": "yellow iris",  
17  "16": "globe-flower",  
18  "17": "purple coneflower",  
19  "18": "peruvian lily",  
20  "19": "balloon flower",  
21  "20": "giant white arum lily"  
22 }  
23
```

*Figure 5: `flowers_to_name.json` with dictionary of numbers and species of flowers*

### 3.4 Learning Model and Transfer Learning

PyTorch enables us to download a specific learning model architecture which we will use for the training through the `torchvision.models` module. However, there are multiple classification models available within the PyTorch framework (i.e. Resnet50, VGG, MobileNetV2, AlexNet etc.)

We employed VGG16 model for this task which has been previously trained on the ImageNet1k dataset containing 1000 classes. In this way, we apply the principles of transfer learning which is a very useful technique in machine learning. Its architecture can be seen in Figure 6.

In transfer learning, the knowledge of an already trained machine learning model is applied to a different but related problem. As the VGG16 model has been trained on a very large dataset, we can use its weights in order to improve the learning and generalization of the features of our present set of images of flowers. This technique is even more useful in our case where the training set is not very large to begin with and the model will benefit from prior knowledge as it doesn't have such a large new dataset from which to learn from scratch.

```
[ ] # Build and train your network
# Transfer Learning
model = models.vgg16(pretrained=True)
model

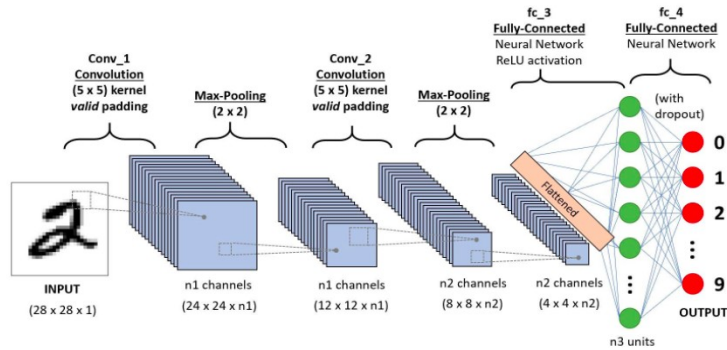
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be removed in the future, please use 'weights' instead.
warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weight enum or 'None' for 'weights' are deprecated since 0.13 and may be removed in the future. The current behavi
warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.cache/torch/hub/checkpoints/vgg16-397923af.pth
100% |#####| 520M/520M [00:02<00:00, 265MB/s]
VGG16
VGG16 (features): Sequential(
  (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): ReLU(inplace=True)
  (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (3): ReLU(inplace=True)
  (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (6): ReLU(inplace=True)
  (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (8): ReLU(inplace=True)
  (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (11): ReLU(inplace=True)
  (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (13): ReLU(inplace=True)
  (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (15): ReLU(inplace=True)
  (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (18): ReLU(inplace=True)
  (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (20): ReLU(inplace=True)
  (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (22): ReLU(inplace=True)
  (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (25): ReLU(inplace=True)
  (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (27): ReLU(inplace=True)
  (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (29): ReLU(inplace=True)
  (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
(avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
(classifier): Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
  (3): Linear(in_features=4096, out_features=4096, bias=True)
  (4): ReLU(inplace=True)
  (5): Dropout(p=0.5, inplace=False)
  (6): Linear(in_features=4096, out_features=1000, bias=True)
)
```

Figure 6: Architecture of the VGG16 model

- **Conv2D Layer:** Two-dimensional convolution is applied over the input fed into the neural network where the specific shape of the input is given in the form of size, length, width, channels.
- **ReLU:** Rectified Linear Unit layer – is an activation function, or a rectifier function, which returns 0 if the input value is less than 0 and it returns the input value otherwise. The reason why the rectifier function is typically used as the activation function in a convolutional neural network is to increase the nonlinearity of the data set. You can think of this as the desire for an image to be as close to gray-and-white as possible. By removing negative values from the neurons' input signals, the rectifier function is effectively removing black pixels from the image and replacing them with gray pixels.
- **MaxPool2d:** The purpose of max pooling it to teach the convolutional neural networks to detect features in an image when the feature is presented in any manner (i.e. closer or further away, partly obstructed by other objects, different colors etc.). This layer helps the neural network to transform a larger feature map into a smaller one, in order to reduce dimensional and computational complexity of the model. The smaller feature map is obtained by overlaying another matrix and extracting the maximum value and composing the smaller feature map with those extracted values.
- **Dropout Layer:** a random selection of neurons is ignored at this step, which forces the neurons to stabilize the network by substituting for the missing ones. In this way, the weight of the neurons do not provide too much information in order for the network to be influenced with very specific features of the training data and become prone to overfitting and impossibility to generalize.

A more general illustration of a convolutional neural networks architecture is presented in Figure 7.

Figure 7: CNN Architecture



Because this model was initially trained on 1000 classes, we need to freeze the initial classification hyper-parameters to make them match our needs for classifying only 20 classes this time. Therefore, we override the classifier variable and define it so it outputs only 20 classes. Because the size of our dataset is significantly smaller than the initial dataset that has been used for training, we add a dropout layer to the architecture so that we ignore a number of neurons so that their weights do not influence the model's capacity to generalize features. In this way we avoid the problem of overfitting, which means that the model has learned the features of the training set so well, it cannot generalize them enough and the accuracy on the testing set will be low.

### 3.5 Training

The training takes place in the `train_classifier()` function.

The number of epochs is set to 15 for this example, but this is a hyper-parameter that needs continuous tuning.

We can leverage again different functions made available by PyTorch such as `model.train()` which helps us to train the images and labels of the training set we loaded in the beginning.

The loss function measures how well the the neural network is performing and its value should decrease during backpropagation in order to make the model work better.

The loss function in our case is implemented from PyTorch: `criterion = nn.NLLLoss()`

In short, the loss is calculated by cross-validating the two parameters of the `NLLLoss()` function (i.e. output and labels), the predicted output with the actual labels we know to be correct. Then, backpropagation is the practice of fine-tuning the weights of a neural network based on the error rate (i.e. loss) obtained in the previous epoch (i.e. iteration). Proper tuning of the weights ensures lower error rates, making the model reliable by increasing its generalization.

The backpropagation step is done by feeding the calculated loss back into the neural network as follows: `loss.backward()`

For the optimization step, `optimizer = optim.Adam(model.classifier.parameters(), lr=0.001)` is the chosen gradient descent optimizer in order to adapt the learning rate of each weight in the neural network.

All these elements can be seen in Figure 8.

```

# Train the classifier
from workspace_utils import active_session

def train_classifier():
    with active_session():
        epochs = 15
        steps = 0
        print_every = 40
        model.to('cuda')

        for e in range(epochs):
            model.train()

            running_loss = 0

            for images, labels in iter(train_loader):
                steps += 1

                images, labels = images.to('cuda'), labels.to('cuda')

                optimizer.zero_grad()

                output = model.forward(images)
                loss = criterion(output, labels)
                loss.backward()
                optimizer.step()

                running_loss += loss.item()

            if steps % print_every == 0:
                model.eval()

                # Turn off gradients for validation, saves memory and computations
                with torch.no_grad():
                    validation_loss, accuracy = validation(model, validate_loader, criterion)

                print("Epoch: {}/{}.. ".format(e+1, epochs),
                      "Training Loss: {:.3f}.. ".format(running_loss/print_every),
                      "Validation Loss: {:.3f}.. ".format(validation_loss/len(validate_loader)),
                      "Validation Accuracy: {:.3f}%".format(accuracy/len(validate_loader)))

                running_loss = 0
                model.train()

            train_classifier()

```

Figure 8: Training function

In Figure 9 we can see the values of the loss for each set of data

- Training Loss: indicates how well the model is fitting the training data
- Validation Loss: indicates how well the model fits new data
- Validation Accuracy: how well the model is able to generalize

```

Epoch: 3/15.. Training Loss: 0.882.. Validation Loss: 0.242.. Validation Accuracy: 0.913
Epoch: 6/15.. Training Loss: 0.826.. Validation Loss: 0.317.. Validation Accuracy: 0.938
Epoch: 8/15.. Training Loss: 0.345.. Validation Loss: 0.273.. Validation Accuracy: 0.961
Epoch: 11/15.. Training Loss: 0.878.. Validation Loss: 0.287.. Validation Accuracy: 0.914
Epoch: 14/15.. Training Loss: 0.836.. Validation Loss: 0.382.. Validation Accuracy: 0.929

```

Figure 9: Loss Values on training and validation sets and Validation Accuracy across epochs (iterations)

### 3.6 Validation

The validation step is important to observe how the prediction is improving based on the tuning of the hyper-parameters (Figure 10).

```

# Function for the validation pass
def validation(model, validate_loader, criterion):
    val_loss = 0
    accuracy = 0

    for images, labels in iter(validate_loader):
        images, labels = images.to('cuda'), labels.to('cuda')

        output = model.forward(images)
        val_loss = criterion(output, labels).item()

        probabilities = torch.exp(output)
        equality = (labels.data == probabilities.max(dim=1)[1])
        accuracy += equality.type(torch.FloatTensor).mean()

    return val_loss, accuracy

```

Figure 10: Validation function

### 3.7 Testing

The final part of the example is testing our neural network. In Figure 11 we can see that the overall testing accuracy is 88%



```

def test_accuracy(model, test_loader):
    # Do validation on the test set
    model.eval()
    model.to('cuda')

    with torch.no_grad():
        accuracy = 0

        for images, labels in iter(test_loader):
            images, labels = images.to('cuda'), labels.to('cuda')

            output = model.forward(images)
            probabilities = torch.exp(output)
            equality = (labels.data == probabilities.max(dim=1)[1])
            accuracy += equality.type(torch.FloatTensor).mean()

        print("Test Accuracy: {}".format(accuracy/len(test_loader)))

test_accuracy(model, test_loader)

```

Figure 11: Testing Accuracy

In the case of sample prediction, we can see an example in Figure 12 and 13.

The image has been classified as part of class 15 and the illustration shows the flower and the graphical representation of the class it belongs to.

```

# Implement the code to predict the class from an image file
def predict(image_path, model, topk=5):
    """ Predict the class (or classes) of an image using a trained deep learning model.
    """

    image = process_image(image_path)

    # Convert image to PyTorch tensor first
    image = torch.from_numpy(image).type(torch.cuda.FloatTensor)
    #print(image.shape)
    #print(type(image))

    # Returns a new tensor with a dimension of size one inserted at the specified position.
    image = image.unsqueeze(0)

    output = model.forward(image)
    probabilities = torch.exp(output)

    # Probabilities and the indices of those probabilities corresponding to the classes
    top_probabilities, top_indices = probabilities.topk(topk)

    # Convert to lists
    top_probabilities = top_probabilities.detach().type(torch.FloatTensor).numpy().tolist()[0]
    top_indices = top_indices.detach().type(torch.FloatTensor).numpy().tolist()[0]

    # Convert topk indices to the actual class labels using class_to_idx
    # Invert the dictionary so you get a mapping from index to class.
    idx_to_class = {value: key for key, value in model.class_to_idx.items()}
    #print(idx_to_class)

    top_classes = [idx_to_class[index] for index in top_indices]

    return top_probabilities, top_classes

probs, classes = predict('flowers/test/15/image_06369.jpg', model)
print(probs)
print(classes)

```

Figure 12: The classes the sample belongs to with each probability

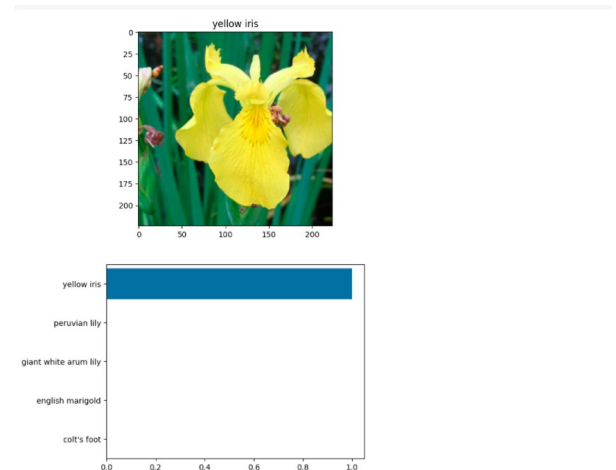


Figure 13: The image and the species it belongs to as graphical representation

## Conclusion

Overall, PyTorch provides an excellent framework and platform for researchers and developers to work on cutting-edge deep learning problems while focusing on the tasks that matter and be able to easily debug, experiment, and deploy.