IMAGE CLASSIFICATION WITH PYTORCH & CNN | TOWARDS AI

Image Classification using Deep Learning & PyTorch: A Case Study with Flower Image Data

Classifying Flower images using Convolutional Deep Neural Network with PyTorch library

Avishek Nag Follow
Aug 5, 2019 · 8 min read ★

Classifying image data is one of the very popular usages of Deep Learning techniques. In this article, we will discuss the identification of flower images using a deep convolutional neural network.

For this, we will be using PyTorch, TorchVision & PIL libraries of Python

Data Exploration

The required dataset for this problem can be found at Kaggle. It contains a folder structure & flower images inside it. There are 5 different types of flowers. The folder structure looks like below

Fig 1

Now, we will see a sample of the flower image from folder 'rose'

```
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt

def show_image(path):
    img = Image.open(path)
    img_arr = np.array(rose_img)
    plt.figure(figsize=(5,5))
    plt.imshow(np.transpose(img_arr, (0, 1, 2)))
```

```
show_image("../data/flowers/rose/537207677_f96a0507bb.jpg")
```

Data Pre-processing

PyTorch always expects data in the form of 'tensors'. These 'tensors' runs between nodes of Neural Network and contains original & pre or post-processed data. Basically, in short, 'tensors' are analogous to 'numpy' arrays.

For image data also, we have to read the images as tensors and apply several pre-processing stages before doing any sort of classification.

We can consider an image as a three-dimensional tensor. Each image can have 3 types of color values of its pixels — Red, Green & Blue respectively. We call this RGB color coding. The other two dimensions are length & width-wise pixel values.

Generally, two of the very common pre-processing stages are required for image data as given below:

1. **Resize to a template:** Resizing the image to square-shaped. For our case, we will resize each image to a 64x64 image.
2. **Normalization:** Statistical normalization using (x — mean)/sd mechanism of each

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

transformations = transforms.Compose([
    transforms.RandomResizedCrop(64),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

total_dataset = datasets.ImageFolder("../data/flowers", transform = transformations)
dataset_loader = DataLoader(dataset = total_dataset, batch_size = 100)
items = iter(dataset_loader)
image, label = items.next()
```

image_preprocessing.py hosted with ❤️ by GitHub view raw

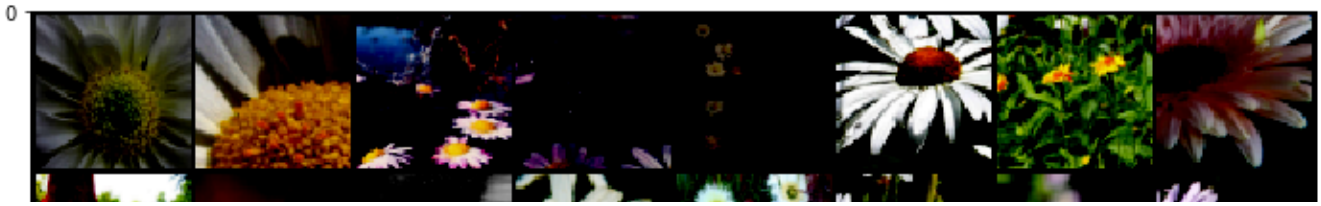
We can also define a function to display a set of transformed images

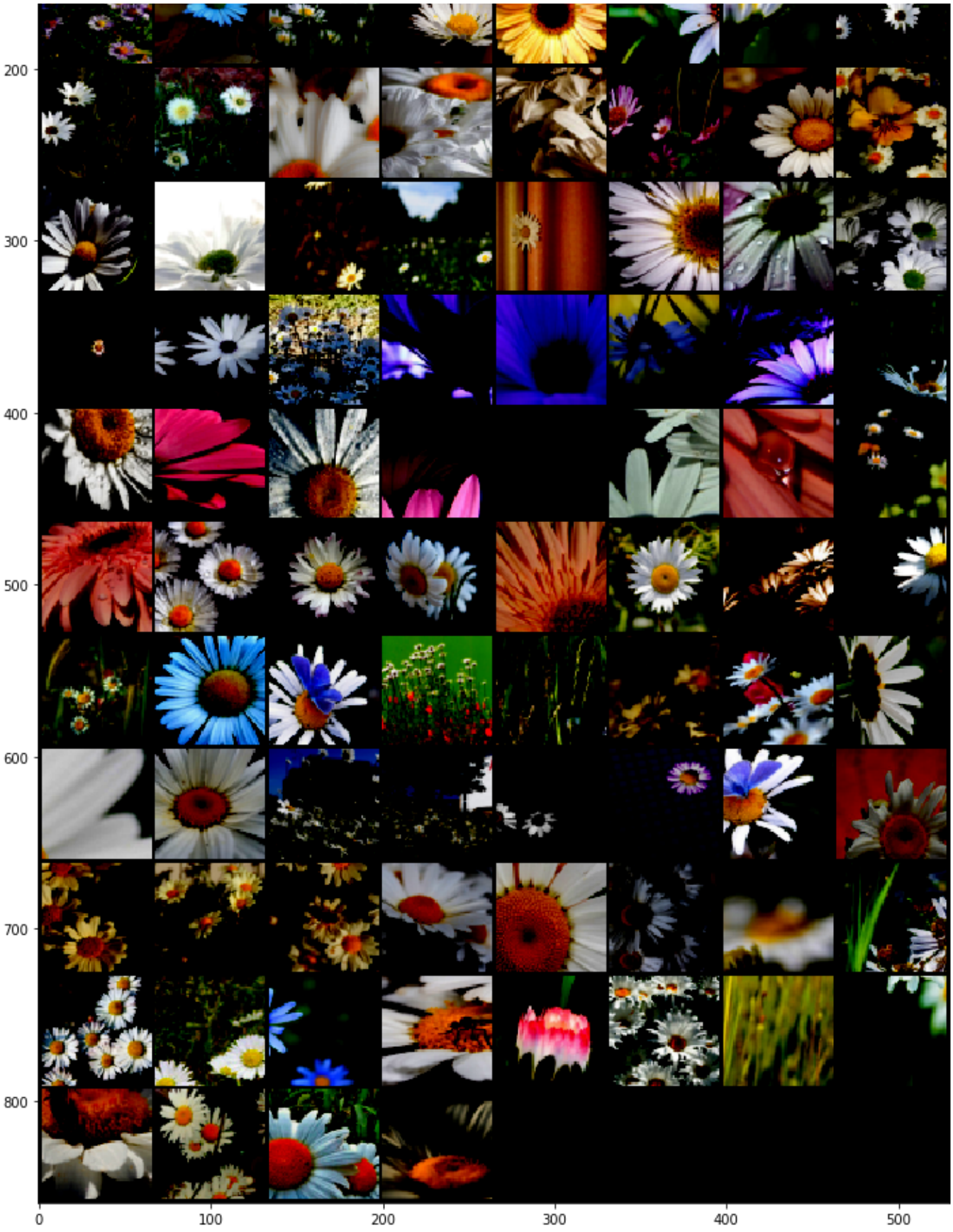
```python
def show_transformed_image(image):
    np_image = image.numpy()
    plt.figure(figsize=(20,20))
    plt.imshow(np.transpose(np_image, (1, 2, 0)))
```

display_transformed_images_grid.py hosted with ❤️ by GitHub view raw

Now, we can see transformed images of the first batch

```
show_transformed_image(make_grid(image))
```




200
300
400
500
600
700
800
0
100
200
300
400
500

```
{'daisy': 0, 'dandelion': 1, 'rose': 2, 'sunflower': 3, 'tulip': 4}
```

This will help in identifying the classes.

Building the Model

To build a machine learning model for image data, just feeding the pixel values is not sufficient. There are many hidden features in the image that will remain undiscovered. For this, we should use a combination of the convolution & max-pooling layer to extract important features.

Convolution layer

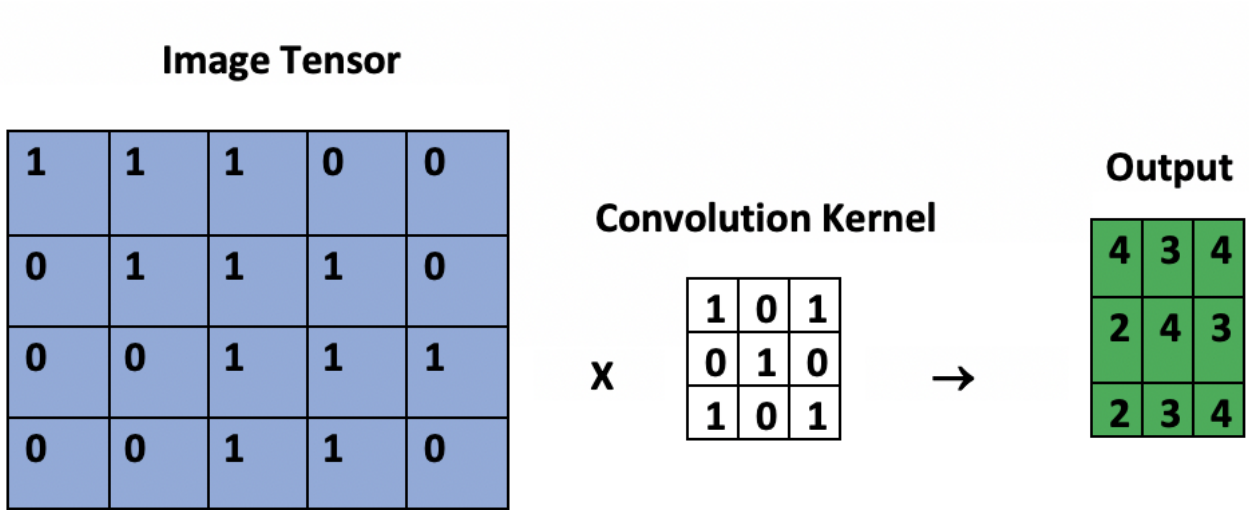
Mathematically, a convolution operation between two functions f & g is defined as

$$f * g = \sum_{a} f(a).g(a)$$

Practically, if we consider f as the image tensor then g should be another tensor that can work as a “convolution kernel”.

> *It is a pixel-wise summation of the multiplicaticed values of two tensors.*

Below diagram shows the effect of the convolution operation on a sample image tensor



Convolution kernel of size 3x3 moves around the image tensor as a window starting from (0,0) location and the sample result at (0,0) of output tensor comes like below calculation

> *Output(0,0) = Image Tensor(0,0) x Kernel(0,0) + Image Tensor(0,1) x Kernel(0,1)+ Image Tensor(0,2) x Kernel(0,2) + Image Tensor(1,0) x Kernel(1,0) + Image Tensor(1,1) x Kernel(1,1) + Image Tensor(1,2) x Kernel(1,2) + Image Tensor(2,0) x Kernel(2,0) + Image Tensor(2,1) x Kernel(2,1) + Image Tensor(2,2) x Kernel(2,2)= 1 x 1 + 1 x 0 + 1 x 1 + 0 x 0 + 1 x 1 + 1 x 0 + 0 x 1 + 0 x 0 + 1 x 1 = 4*

The kernel shifts location by 1 place to calculate values of other locations of the output tensor. This 'shift' is known as 'stride'.

> *Convolution layer is needed to enhance and extract important & hidden features of the image. In our case, it may happen that the 'flower' is located at the central position in the image, so applying convolution helps to retrieve the core features of the flower ignoring other background objects & colors.*

As each image follows RGB color coding, we will apply a convolution operation for each color and hence we will get three output tensors. The final output will be the tensor summation of all three. Each of these 'color code' is known as 'channel' in PyTorch API terminology.

> *Mathematically, if a filter of size kxk is applied on an image of size WxH then it produces an output image/tensor of size (W-k+1)x(H-k+1)*

In our case, convolution is created like this

```
self.conv1 = nn.Conv2d(in_channels=3, out_channels=12,
kernel_size=3,stride=1, padding=1)
```

> *'out_channels' specifies how many filters to be applied. Here we have applied 12 filters and these will produce 12 intermediate image tensors of size 62x62. Each of these images*

Convolutional Neural Networks (CNN): Step 1- Convolution Operation — Blogs SuperDataScience — Big...

Step 1 — Convolution Operation In this tutorial, we are going to learn about convolution, which is the first step in...

www.superdatascience.com

ReLU layer

'ReLU' is an activation function that captures non-linearity in the output of another function. Mathematically, it is defined as

$$f(x) = max(0, x)$$

So, it always returns the positive value. We can say, it is a 'positive filter'. We will apply the 'ReLU' layer after convolution.

In our case, 'ReLU' is created like below

```
self.relu1 = nn.ReLU()
```

Max Pooling layer

The 'Max pooling layer' generally comes after 'ReLU'. A 'max pool' of size 2 is a 2x2 window which traverses the output tensor of 'ReLU' operation and selects maximum pixel value within the window. This operation can be explained by the below diagram

The objective of the 'Max Pool' layer is to select only those features which are high impacting and have a large value. It helps to reduce the dimensions of the features.

In our case, 'Max Pool' is created like below

```
self.maxpool1 = nn.MaxPool2d(kernel_size=2)
```

It will reduce the dimension of the image by 50% (32 = 64/2).

Linear function layer

As the name implies, it is a linear function which takes the output of 'Max Pool' as a flattened array and produces output as class indices. The output value from the 'Linear function' for the predicted class indices will be maximum.

In our case, 'Linear function' is created like below

```
self.lf = nn.Linear(in_features=32 * 32 * 24,
out_features=num_classes)
```

Overall Architecture of the model

We will apply different layers like below diagram



We have two sets of ‘convolution’ & ‘ReLU’ layers. ‘View’ does the flattening of output tensors from the last ‘ReLU’ layer. We have image tensor of size 64x64 as input which will be reduced to 32x32 due to the application of ‘MaxPool2D’ of kernel size 2x2 (32 = 64/2).

First, we will divide the dataset into train & test by 80:20 ratio

```
from torch.utils.data import random_split

train_size = int(0.8 * len(total_dataset))
test_size = len(total_dataset) - train_size
train_dataset, test_dataset = random_split(total_dataset, [train_size, test_size])

train_dataset_loader = DataLoader(dataset = train_dataset, batch_size = 100)
test_dataset_loader = DataLoader(dataset = test_dataset, batch_size = 100)
```

train_test_loader.py hosted with ❤ by GitHub view raw

Then, we will write a custom class to stack these layers by extending ‘Module’ given by PyTorch library

```
import torch.nn as nn

class FlowerClassifierCNNModel(nn.Module):

    def __init__(self, num_classes=5):
        super(FlowerClassifierCNNModel,self).__init__()

        self.conv1 = nn.Conv2d(in_channels=3, out_channels=12, kernel_size=3,stride=1, p
        self.relu1 = nn.ReLU()

```

```
        self.lf = nn.Linear(in_features=32 * 32 * 24, out_features=num_classes)

    def forward(self, input):
        output = self.conv1(input)
        output = self.relu1(output)

        output = self.maxpool1(output)

        output = self.conv2(output)
        output = self.relu2(output)

        output = output.view(-1, 32 * 32 * 24)

        output = self.lf(output)

        return output
```

FlowerImageClassifierNN.py hosted with ❤️ by GitHub view raw

'__init__' defines each layer & its parameters whereas 'forward' function does the actual calls & layer stacking. The output from the final layer is returned from the 'forward' function.

Model training

We need to have an optimizer & loss function for model training. We will use 'Adam optimizer' & 'Cross-Entropy Loss' for this.

```
from torch.optim import Adam

cnn_model = FlowerClassifierCNNModel()
optimizer = Adam(cnn_model.parameters())
loss_fn = nn.CrossEntropyLoss()
```

With PyTorch, we need to set the model in training mode and then run the training by iterating through the training dataset, calculating the loss and incrementing steps of the

```
def train_and_build(n_epoches):
    for epoch in range(n_epoches):
        cnn_model.train()
        for i, (images, labels) in enumerate(train_dataset_loader):
            optimizer.zero_grad()
            outputs = cnn_model(images)
            loss = loss_fn(outputs, labels)
            loss.backward()
            optimizer.step()
```

pytorch_training_cnn_model.py hosted with ❤ by GitHub view raw

The function call 'loss.backward' goes back to the layers and calculates loss occurred during the process.

We will use an 'epoche' of 200 to train the model

```
train_and_build(200)
```

Model testing & Accuracy

We should set the model to 'eval' mode for testing its accuracy on the testing dataset

```
import torch

cnn_model.eval()
test_acc_count = 0
for k, (test_images, test_labels) in enumerate(test_dataset_loader):
    test_outputs = cnn_model(test_images)
    _, prediction = torch.max(test_outputs.data, 1)
    test_acc_count += torch.sum(prediction == test_labels.data).item()

test_accuracy = test_acc_count / len(test_dataset)
```

test_accuracy_cnn_model.py hosted with ❤ by GitHub view raw

> *'torch.sum' function sums up the '1's present in tensor which is the output of 'AND' operation between 'predicted' & 'actual test output' tensor. So this summation gives the no of images predicted correctly.*

And here we get the accuracy

```
test_accuracy
```


It is almost 70.52%. We got good accuracy with a simple model. This model can be further tunned.

Using the model for prediction with a sample image

We will now see how to use this model with a sample image from our dataset.

```
show_image("../data/flowers/dandelion/13920113_f03e867ea7_m.jpg")
```

Now we will read the image using PIL image API and feed it into our transformation pipeline for necessary preprocessing and later use the model for prediction

```
test_image = Image.open("../data/flowers/dandelion/13920113_f03e867ea7_m.jpg")
test_image_tensor = transformations(test_image).float()
test_image_tensor = test_image_tensor.unsqueeze_(0)
output = cnn_model(test_image_tensor)
class_index = output.data.numpy().argmax()
```

image_prediction_using_model.py hosted with ❤ by GitHub view raw

```
class_index
```

So, from class to index dictionary as mentioned above, we can confirm that it is a 'Dandelion'. So, our image classifier model is working well !!

Conclusion

We learned how to do image classification using the PyTorch library. In the process, we covered pre-processing of the image, building the convolution layer and testing the model for an input image.

> *Accuracy of the model can be further improved by 'Hyperparameter' tunning like experimenting with 'Adam optimizer' parameters, adding extra convolution layers, tunning with kernel size & max pool window size, etc. Readers of this article can try these techniques on their own.*

Jupyter notebook for this can be found from the below link

Machine Learning Deep Learning Image Classification Convolutional Neural Net Pytorch

About Help Legal