TRANSFER LEARNING

TUTORIAL

What is "Transfer Learning"?

- "Transfer Learning" can be understood as we use our knowledge in one task and applying it to another different but related task.

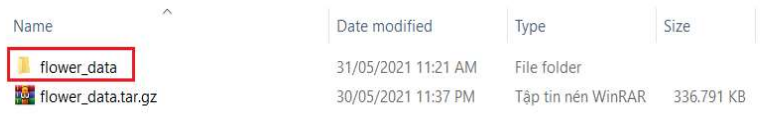
- For example, in computer vision's problems, we use a model obtained the high performance on **truck recognition** and applied it to solve the **car classification** task.

PRACTICE

Now, we will practice to use "Transfer learning" for solving flowers classification problem. we will use a different custom model of VGG-16 Net to solve this problem. The dataset which we use is Oxford Flower 102 dataset.

- **<u>Step 1</u>:** Download, unzip and upload the dataset to Google Drive. Download it here.

  + After we download and unzip downloaded file, we will have a folder like this.



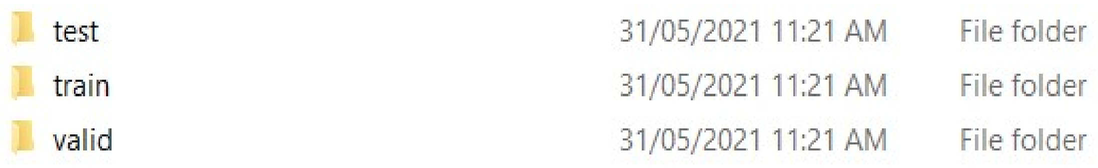
Name	Date modified	Type	Size
flower_data	31/05/2021 11:21 AM	File folder	
flower_data.tar.gz	30/05/2021 11:37 PM	Tập tin nén WinRAR	336.791 KB

  + Upload 'flower_data' folder to Google Drive.

  + Some dataset's information:

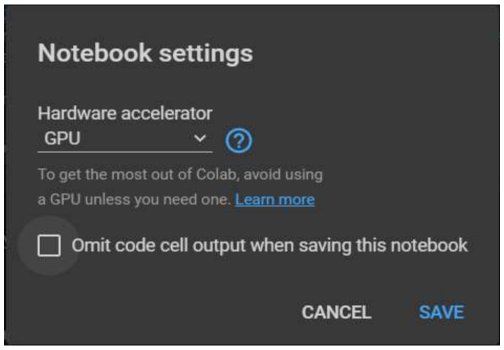
    - This dataset contains images of 102 flower categories commonly occurring in the United Kingdom. Here are some examples.

- This dataset has already splitted to three sets: training, testing and validation.



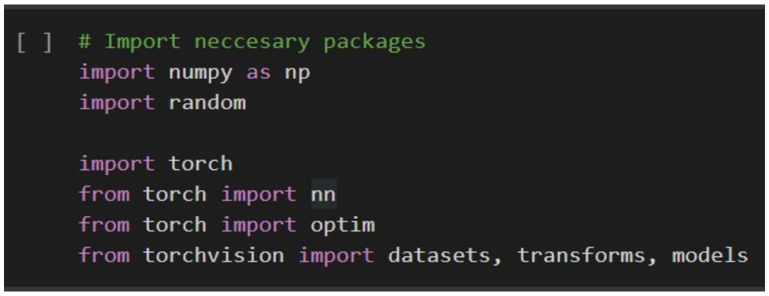
```
test     31/05/2021 11:21 AM    File folder
train    31/05/2021 11:21 AM    File folder
valid    31/05/2021 11:21 AM    File folder
```

- **<u>Step 2</u>:** Open Google Colab, link Google Drive and connect to GPU.



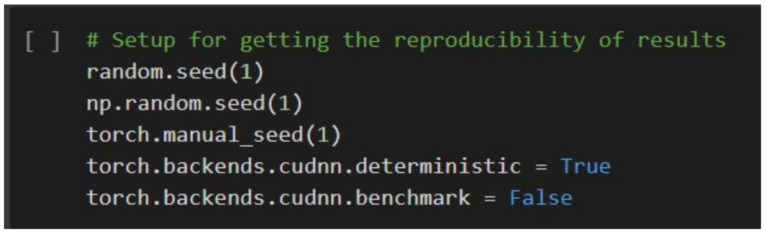
Notebook settings
Hardware accelerator
GPU
To get the most out of Colab, avoid using a GPU unless you need one. Learn more
Omit code cell output when saving this notebook
CANCEL
SAVE

- **<u>Step 3:</u>** Import necessarcy packages.



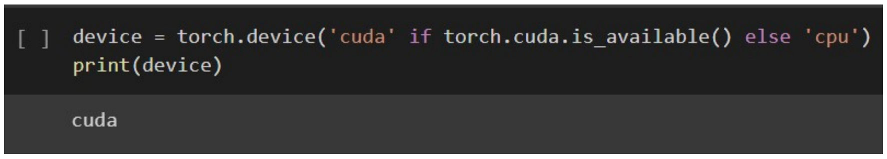
```
# Import neccesary packages
import numpy as np
import random

import torch
from torch import nn
from torch import optim
from torchvision import datasets, transforms, models
```

Next, we setup for getting the reproducible results



```
# Setup for getting the reproducibility of results
random.seed(1)
np.random.seed(1)
torch.manual_seed(1)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
```

Check whether we are using GPU. If the output of 'print(device)' line is 'cuda', we are going on the right way.



```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
```

```
cuda
```

- **Step 4**: Load the dataset

Before loading the dataset, we read some information about VGG-16 Net:

+ VGG-16 Net is a convolutional neural network and was trained on ImageNet dataset.

+ The input of VGG-16 is a color 224 x 224 image.

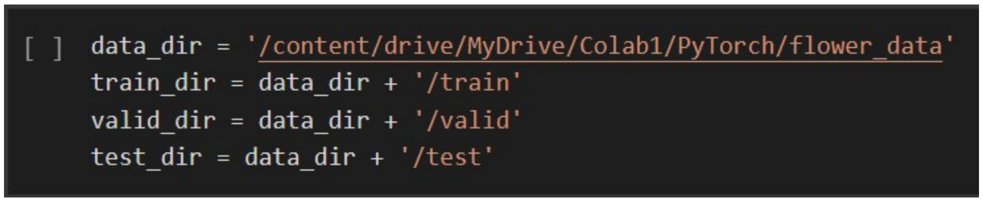
+ VGG-16 Net were trained on the ImageNet dataset where each color channel was normalized separately.

Therefore, to use VGG-16 on Oxford Flower 102 dataset, we have to do below transform steps:

+ Resize input images to 224 x 224 pixel.

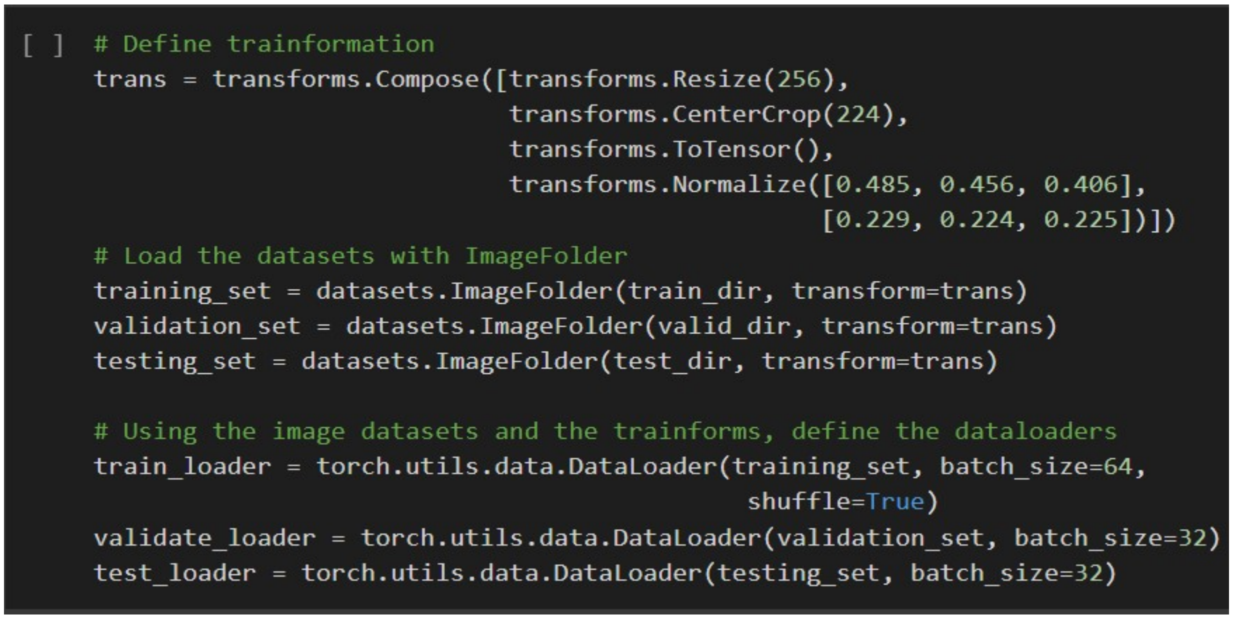
+ Normalize the means and standard deviations of the images.

Here are steps for loading Oxford Flower 102 dataset.



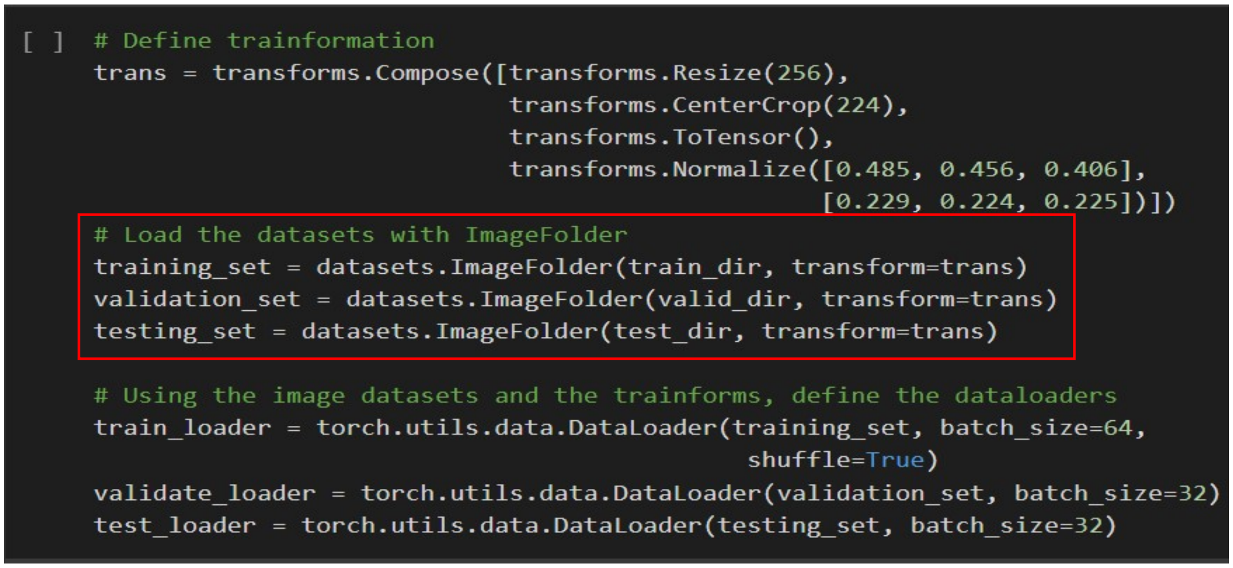
```
[ ]  data_dir = '/content/drive/MyDrive/Colab1/PyTorch/flower_data'
     train_dir = data_dir + '/train'
     valid_dir = data_dir + '/valid'
     test_dir = data_dir + '/test'
```

+ 'data_dir': the path where you store the dataset in your Google Drive



```
[ ]  # Define trainformation
     trans = transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485, 0.456, 0.406],
                                                      [0.229, 0.224, 0.225])])
     # Load the datasets with ImageFolder
     training_set = datasets.ImageFolder(train_dir, transform=trans)
     validation_set = datasets.ImageFolder(valid_dir, transform=trans)
     testing_set = datasets.ImageFolder(test_dir, transform=trans)

     # Using the image datasets and the trainforms, define the dataloaders
     train_loader = torch.utils.data.DataLoader(training_set, batch_size=64,
                                                shuffle=True)
     validate_loader = torch.utils.data.DataLoader(validation_set, batch_size=32)
     test_loader = torch.utils.data.DataLoader(testing_set, batch_size=32)
```

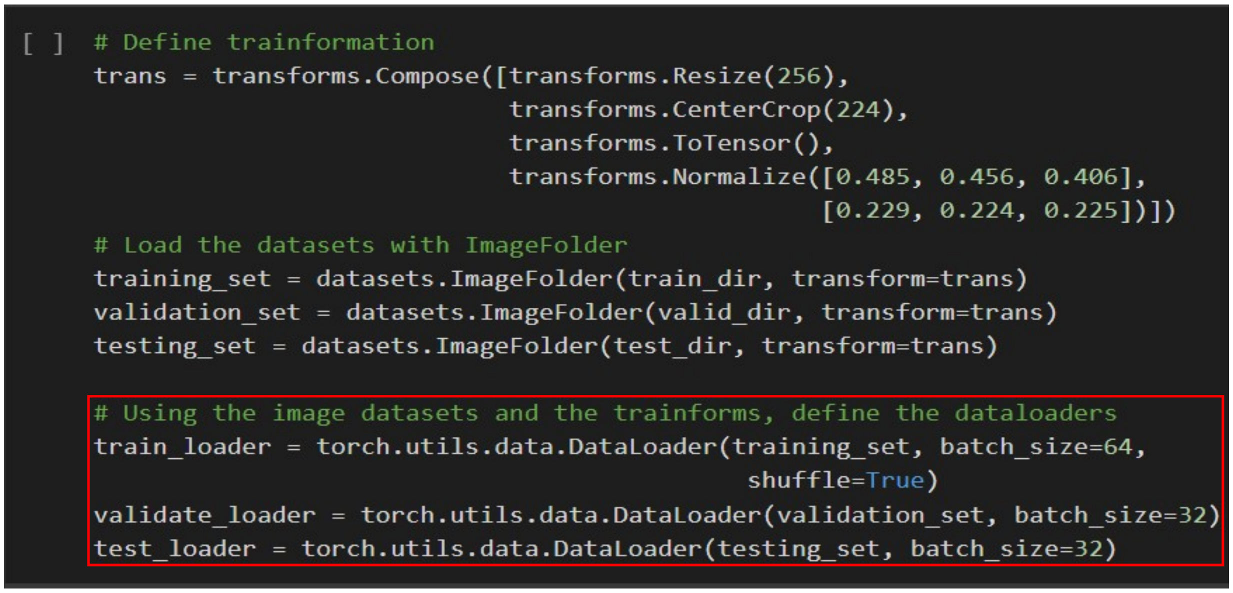
+ We create training, validation, testing set by using dataset.ImageFolder. When creating, we apply the transformation which we discussed above on each image.



```
[ ]  # Define trainformation
     trans = transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485, 0.456, 0.406],
                                                      [0.229, 0.224, 0.225])])
     # Load the datasets with ImageFolder
     training_set = datasets.ImageFolder(train_dir, transform=trans)
     validation_set = datasets.ImageFolder(valid_dir, transform=trans)
     testing_set = datasets.ImageFolder(test_dir, transform=trans)

     # Using the image datasets and the trainforms, define the dataloaders
     train_loader = torch.utils.data.DataLoader(training_set, batch_size=64,
                                                shuffle=True)
     validate_loader = torch.utils.data.DataLoader(validation_set, batch_size=32)
     test_loader = torch.utils.data.DataLoader(testing_set, batch_size=32)
```

In 'tranforms' variable, the values [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225] are means and standard deviations of the images on ImageNet dataset. Besides, we resize image by resize it to 256 x 256 image (tranform.Resize(256)) and crop it to a 224 x 224 image then (transform.CenterCrop(224)). You can try other tranformations to resize image to 224 x 224.

+ After creating training, validation, testing set, we use DataLoader to split data to small batches.



```
[ ]  # Define trainformation
     trans = transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize([0.485, 0.456, 0.406],
                                                      [0.229, 0.224, 0.225])])
     # Load the datasets with ImageFolder
     training_set = datasets.ImageFolder(train_dir, transform=trans)
     validation_set = datasets.ImageFolder(valid_dir, transform=trans)
     testing_set = datasets.ImageFolder(test_dir, transform=trans)

     # Using the image datasets and the trainforms, define the dataloaders
     train_loader = torch.utils.data.DataLoader(training_set, batch_size=64,
                                                shuffle=True)
     validate_loader = torch.utils.data.DataLoader(validation_set, batch_size=32)
     test_loader = torch.utils.data.DataLoader(testing_set, batch_size=32)
```

- **Step 5**: Load model (VGG-16)

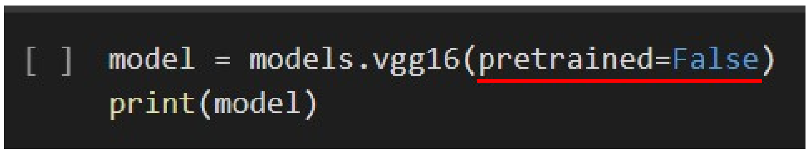
At this step, we will try 2 different approaches.

+ The first approach is that we customize the classifer of VGG-16 Net but do not use the parameters of pre-trained model and train the model from scratch (we update parameters of all layers).

+ The second approach is that we customize the classifer of VGG-16 but use the parameters of pre-trained model and just perform parameters updating on the classifer.

5.1. The first approach

+ Load VGG-16 model (no using pre-trained model)

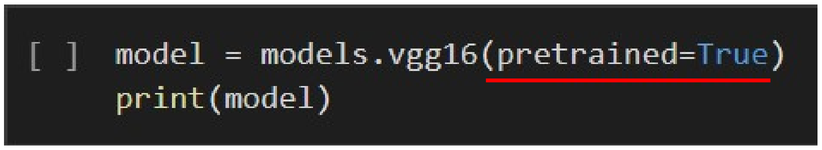


```
[ ]  model = models.vgg16(pretrained=False)
     print(model)
```

→ Set 'pretrained' value to 'True' if we want to load pretrained model.

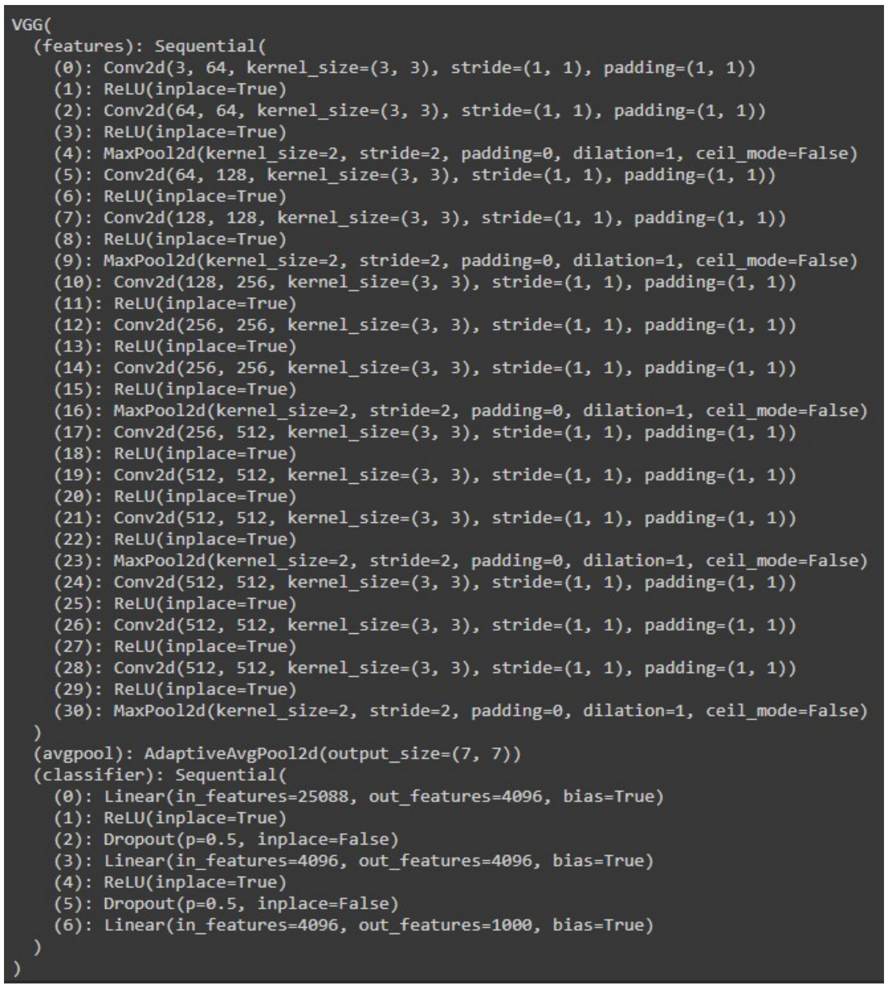
5.2. The second approach

+ Load VGG-16 model (using pre-trained)



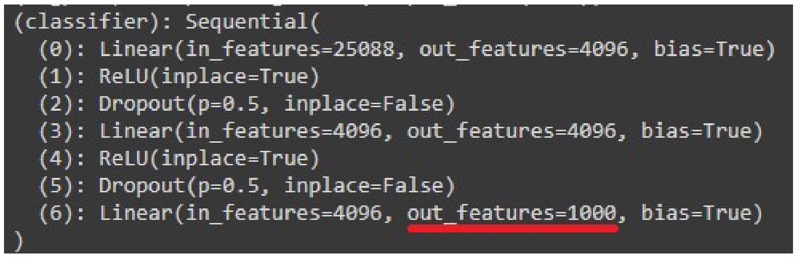
```
[ ]  model = models.vgg16(pretrained=True)
     print(model)
```

Here is the architecture of VGG-16 Net.



```
VGG(
  (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)
  )
)
```

We can see in the ‘classifier’ of VGG-16, the output is 1 of out 1000 labels. But in Oxford Flower dataset, the number of categories is 102. Thus, we need to customize the classifier.



```
(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)
)
```

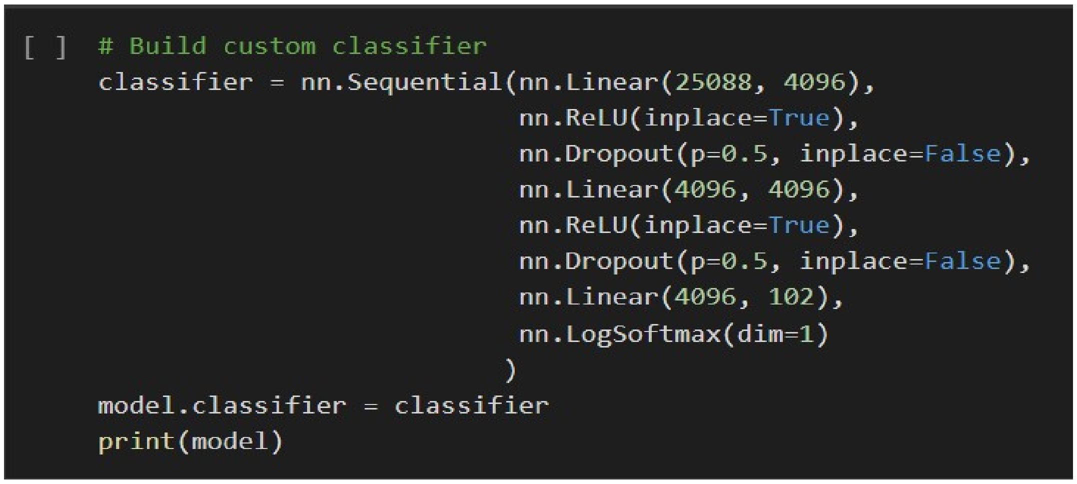
- **<u>Step 6</u>**: Build new classifier

We build a new classifier by change the value of ‘out_features’ to 102 and add ‘LogSoftmax’ layer to classify. After a new classifier was built, replace the current classifier of model by it.

<u>Note:</u>

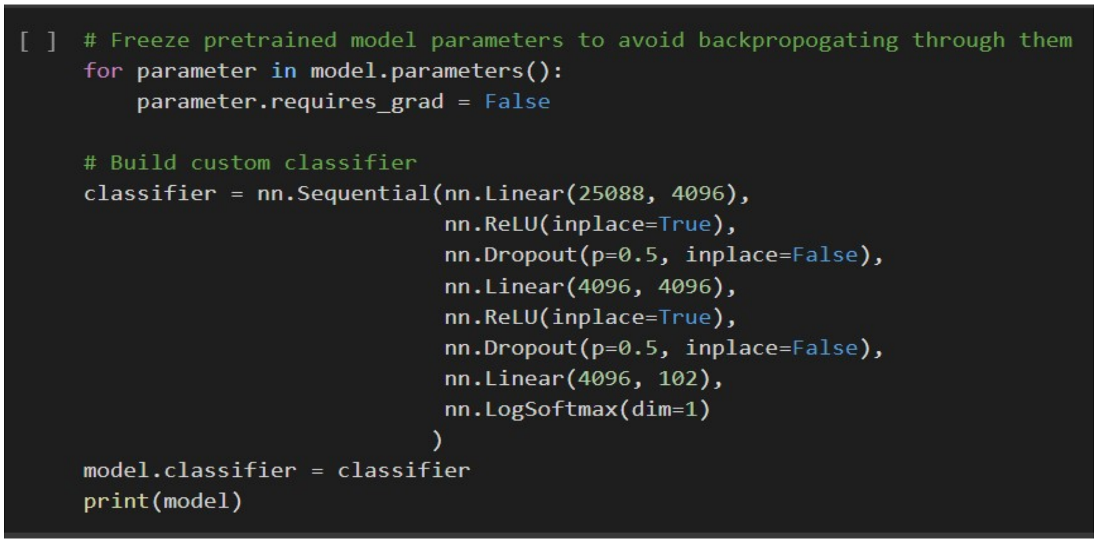
In the second approach, before replacing the classifier, we must to freeze the parameters of pre-trained model to avoid backpropogating through them.

6.1. The first approach



```
[ ]  # Build custom classifier
     classifier = nn.Sequential(nn.Linear(25088, 4096),
                                nn.ReLU(inplace=True),
                                nn.Dropout(p=0.5, inplace=False),
                                nn.Linear(4096, 4096),
                                nn.ReLU(inplace=True),
                                nn.Dropout(p=0.5, inplace=False),
                                nn.Linear(4096, 102),
                                nn.LogSoftmax(dim=1)
                               )
     model.classifier = classifier
     print(model)
```

6.2. The second approach



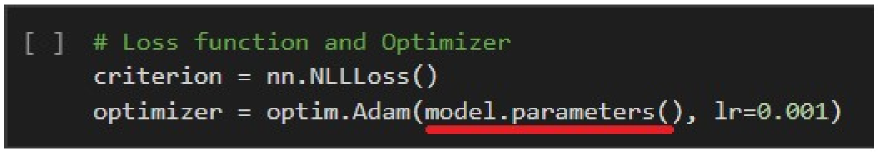
```
[ ]  # Freeze pretrained model parameters to avoid backpropogating through them
     for parameter in model.parameters():
         parameter.requires_grad = False

     # Build custom classifier
     classifier = nn.Sequential(nn.Linear(25088, 4096),
                                nn.ReLU(inplace=True),
                                nn.Dropout(p=0.5, inplace=False),
                                nn.Linear(4096, 4096),
                                nn.ReLU(inplace=True),
                                nn.Dropout(p=0.5, inplace=False),
                                nn.Linear(4096, 102),
                                nn.LogSoftmax(dim=1)
                               )
     model.classifier = classifier
     print(model)
```

- **<u>Step 7</u>**: Define 'loss function' and 'optimizer'

In this experiment, we use 'Negative log likelihood loss' as loss function and 'Adam' as the optimizer.

Even though we use the same optimizer in both approaches, there is still a difference. On the first approach we train model from scratch. Therefore, the optimizer will update to parameters of all layers. On the other hand, we just use optimizer for updating parameters in the classifier on the second approach.

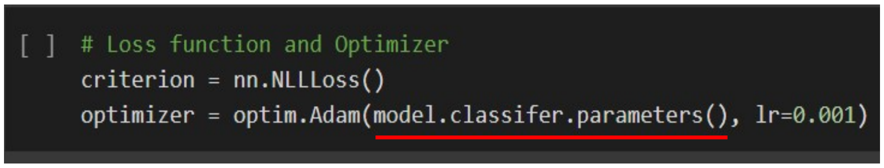
7.1. The first approach



```
[ ]  # Loss function and Optimizer
     criterion = nn.NLLLoss()
     optimizer = optim.Adam(model.parameters(), lr=0.001)
```

→ Learning rate is 0.001

7.2. The second approach



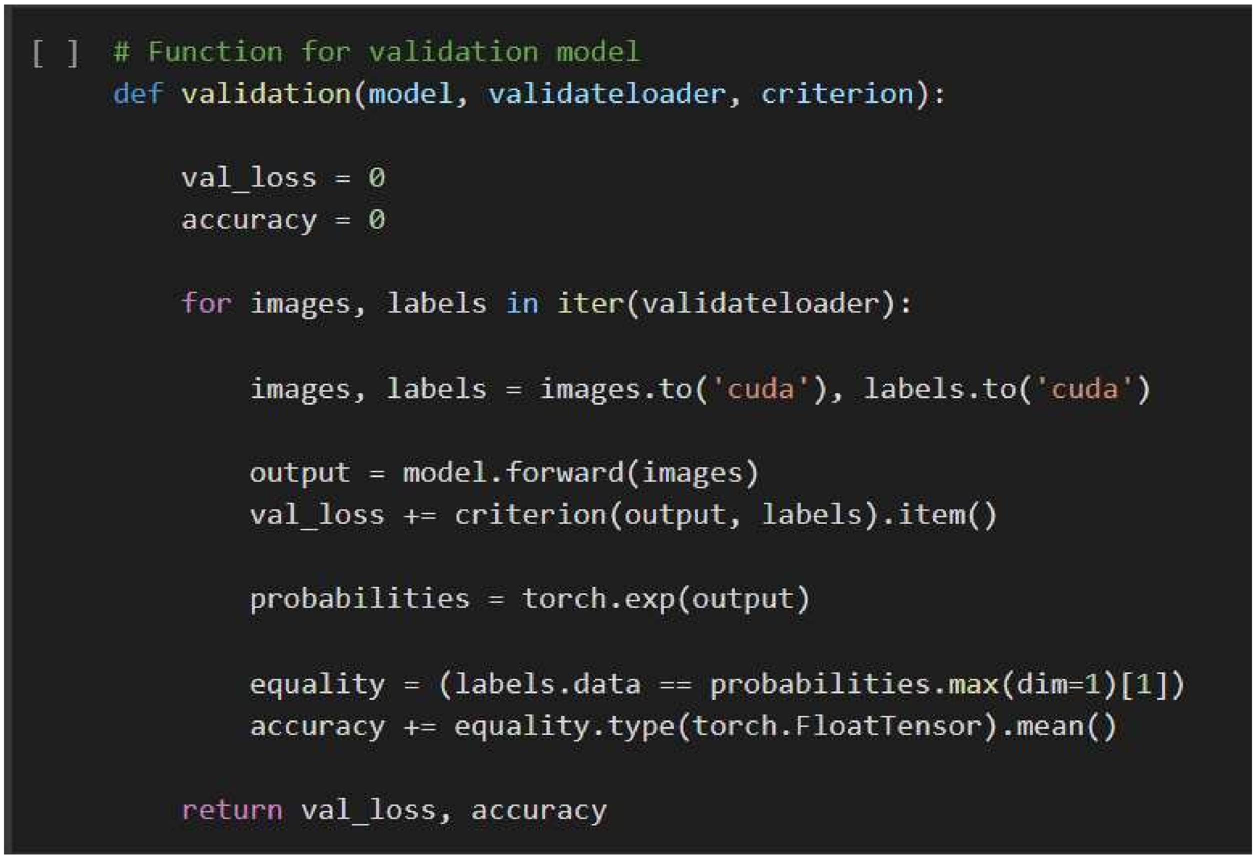
```
[ ]  # Loss function and Optimizer
     criterion = nn.NLLLoss()
     optimizer = optim.Adam(model.classifer.parameters(), lr=0.001)
```

- **Step 8**: Train model

At this step, we will define 2 functions. One function for training model, the other for validating model.

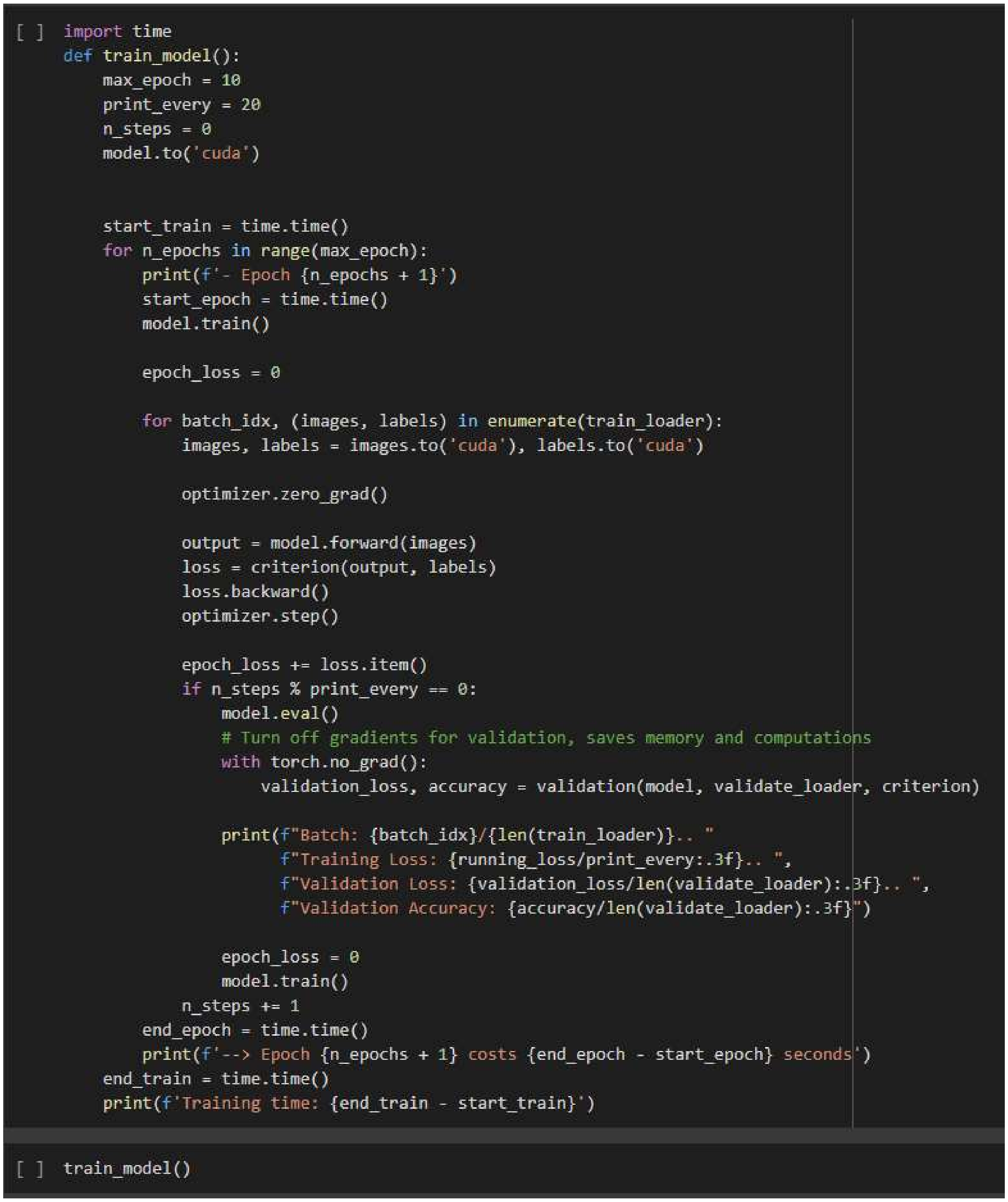
In training processing, we perform model validation each 20 steps for tracking the performance of model. Besides, when validating model, we have to turn off gradient to avoid update parameters.

+ Function for validating model



```
[ ]  # Function for validation model
     def validation(model, validateloader, criterion):

         val_loss = 0
         accuracy = 0

         for images, labels in iter(validateloader):

             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
```

+ Function for training model



```
[ ] import time
    def train_model():
        max_epoch = 10
        print_every = 20
        n_steps = 0
        model.to('cuda')


        start_train = time.time()
        for n_epochs in range(max_epoch):
            print(f'- Epoch {n_epochs + 1}')
            start_epoch = time.time()
            model.train()

            epoch_loss = 0

            for batch_idx, (images, labels) in enumerate(train_loader):
                images, labels = images.to('cuda'), labels.to('cuda')

                optimizer.zero_grad()

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

                epoch_loss += loss.item()
                if n_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(f"Batch: {batch_idx}/{len(train_loader)}.. "
                          f"Training Loss: {running_loss/print_every:.3f}.. ",
                          f"Validation Loss: {validation_loss/len(validate_loader):.3f}.. ",
                          f"Validation Accuracy: {accuracy/len(validate_loader):.3f}")

                    epoch_loss = 0
                    model.train()
                n_steps += 1
            end_epoch = time.time()
            print(f'--> Epoch {n_epochs + 1} costs {end_epoch - start_epoch} seconds')
        end_train = time.time()
        print(f'Training time: {end_train - start_train}')
```

```
[ ] train_model()
```

Okay, now, we compare the training time between both approaches. I don't know why the first epoch took a lot of time but we can consider the other epochs.

If you have trained both approaches, you can see the model in the second approach (use pre-trained model) took less time than the model in the first approach for each trained epoch. The reason is because in the second approach, we have just only updated the parameters in the classifier. But in the first approach, we have to update parameters of entire model.

Some informations:

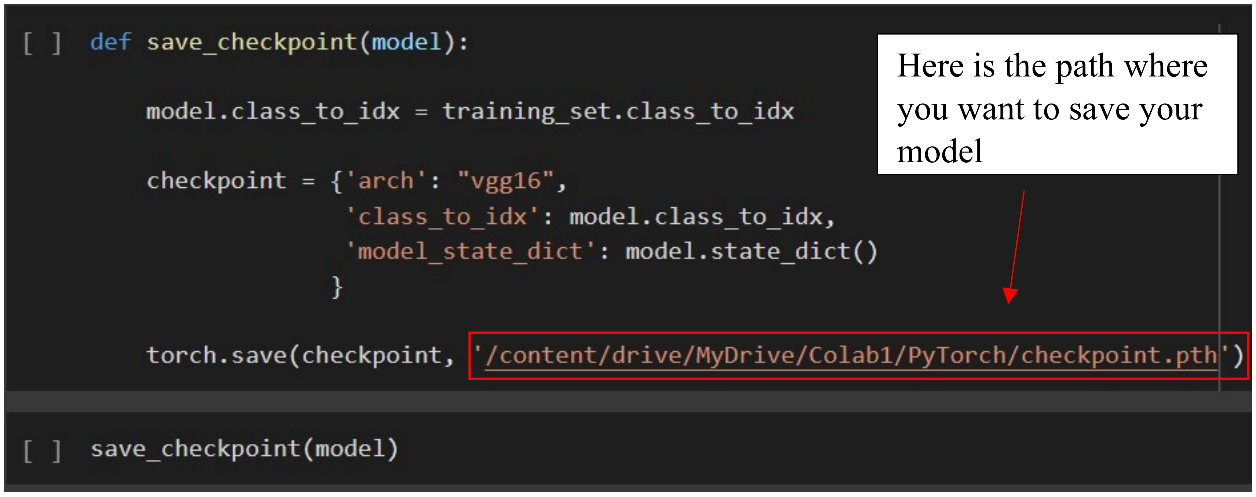
In my experiments:

+ The first appoarch took ≈ 276 seconds for each epoch and ≈ 4524 seconds for training model.

+ The second appoarch took ≈ 178 seconds for each epoch and ≈ 3682 seconds for training model.

- **Step 9:** Save model

We save model for using when we want to use in the later.

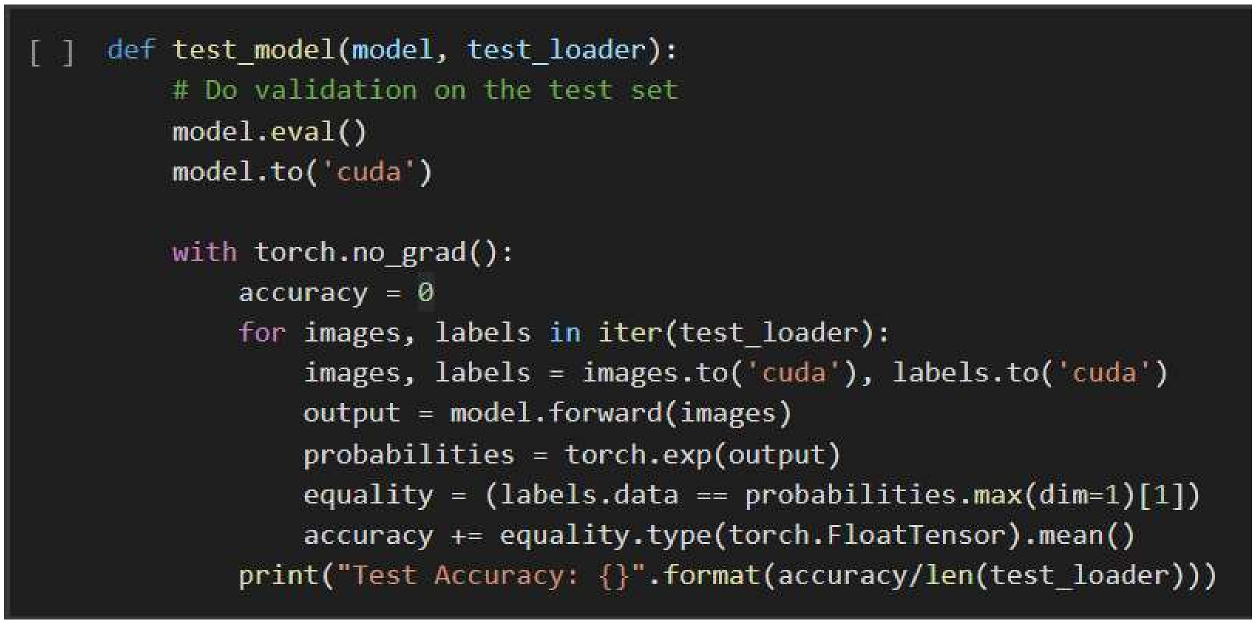


```
[ ] def save_checkpoint(model):

        model.class_to_idx = training_set.class_to_idx

        checkpoint = {'arch': "vgg16",
                      'class_to_idx': model.class_to_idx,
                      'model_state_dict': model.state_dict()
                     }

        torch.save(checkpoint, '/content/drive/MyDrive/Colab1/PyTorch/checkpoint.pth')
```

```
[ ] save_checkpoint(model)
```

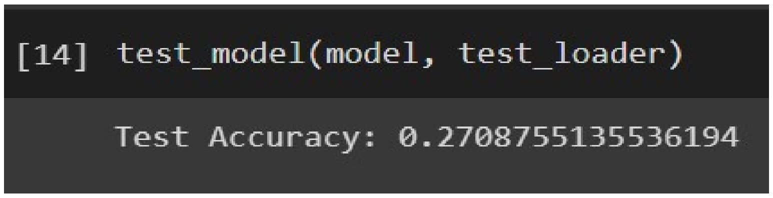
- **Step 10**: Test model

We define a function for testing model



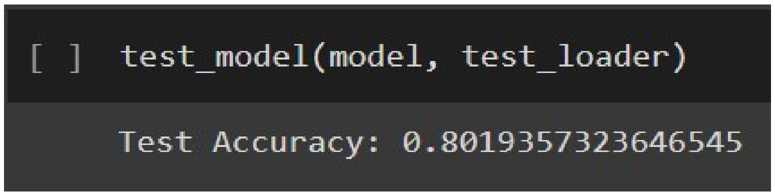
```
[ ]  def test_model(model, test_loader):
         # Do validation on the test set
         model.eval()
         model.to('cuda')

         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)))
```

Performance of model in the first approach (train from scratch)



```
[14] test_model(model, test_loader)

     Test Accuracy: 0.2708755135536194
```

Performance of model in the second approach (use pre-trained model)



```
[ ]  test_model(model, test_loader)

     Test Accuracy: 0.8019357323646545
```

We can see the advantages of using pre-trained model. We have just spent a less time but we can achieve the model which is better than many times the model was trained from scratch.