



7. Assessment

Congratulations on going through today's course! Hopefully, you've learned some valuable skills along the way and had fun doing it. Now it's time to put those skills to the test. In this assessment, you will train a new model that is able to recognize fresh and rotten fruit. You will need to get the model to a validation accuracy of 92% in order to pass the assessment, though we challenge you to do even better if you can. You will have the use the skills that you learned in the previous exercises. Specifically, we suggest using some combination of transfer learning, data augmentation, and fine tuning. Once you have trained the model to be at least 92% accurate on the validation dataset, save your model, and then assess its accuracy. Let's get started!

```
In [34]: import torch
import torch.nn as nn
from torch.optim import Adam
from torch.utils.data import Dataset, DataLoader
import torchvision.transforms.v2 as transforms
import torchvision.io as tv_io

import glob
from PIL import Image

import utils


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

Out[34]: True

7.1 The Dataset

In this exercise, you will train a model to recognize fresh and rotten fruits. The dataset comes from [Kaggle](#), a great place to go if you're interested in starting a project after this class. The dataset structure is in the `data/fruits` folder. There are 6 categories of fruits: fresh

apples, fresh oranges, fresh bananas, rotten apples, rotten oranges, and rotten bananas. This will mean that your model will require an output layer of 6 neurons to do the categorization successfully. You'll also need to compile the model with `categorical_crossentropy`, as we have more than two categories.

 No description has been provided for this image

7.2 Load ImageNet Base Model

We encourage you to start with a model pretrained on ImageNet. Load the model with the correct weights. Because these pictures are in color, there will be three channels for red, green, and blue. We've filled in the input shape for you. If you need a reference for setting up the pretrained model, please take a look at [notebook 05b](#) where we implemented transfer learning.

```
In [35]: from torchvision.models import vgg16
         from torchvision.models import VGG16_Weights

weights = VGG16_Weights.DEFAULT
vgg_model = vgg16(weights=weights)
```

7.3 Freeze Base Model

Next, we suggest freezing the base model, as done in [notebook 05b](#). This is done so that all the learning from the ImageNet dataset does not get destroyed in the initial training.

```
In [36]: # Freeze base model
vgg_model.requires_grad_(False)
next(iter(vgg_model.parameters())).requires_grad
```

Out[36]: False

7.4 Add Layers to Model

Now it's time to add layers to the pretrained model. [Notebook 05b](#) can be used as a guide. Pay close attention to the last dense layer and make sure it has the correct number of neurons to classify the different types of fruit.

The later layers of a model become more specific to the data the model trained on. Since we want the more general learnings from VGG, we can select parts of it, like so:

```
In [37]: vgg_model.classifier[0:3]
```

```
Out[37]: Sequential(
  (0): Linear(in_features=25088, out_features=4096, bias=True)
  (1): ReLU(inplace=True)
  (2): Dropout(p=0.5, inplace=False)
)
```

Once we've taken what we've wanted from VGG16, we can then add our own modifications. No matter what additional modules we add, we still need to end with one value for each output.

```
In [38]: N_CLASSES = 6

my_model = nn.Sequential(
    vgg_model.features,
    vgg_model.avgpool,
    nn.Flatten(),
    vgg_model.classifier[0:3],
    nn.Linear(4096, 500),
    nn.ReLU(),
    nn.Linear(500, N_CLASSES)
)
my_model
```

```
Out[38]: Sequential(
  (0): 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)
  )
  (1): AdaptiveAvgPool2d(output_size=(7, 7))
  (2): Flatten(start_dim=1, end_dim=-1)
  (3): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
```

```
)  
(4): Linear(in_features=4096, out_features=500, bias=True)  
(5): ReLU()  
(6): Linear(in_features=500, out_features=6, bias=True)  
)
```

7.5 Compile Model

Now it's time to compile the model with loss and metrics options. We have 6 classes, so which loss function should we use?

```
In [39]: loss_function = nn.CrossEntropyLoss()  
optimizer = Adam(my_model.parameters())  
my_model = torch.compile(my_model.to(device))
```

7.6 Data Transforms

To preprocess our input images, we will use the transforms included with the VGG16 weights.

```
In [40]: pre_trans = weights.transforms()
```

Try to randomly augment the data to improve the dataset. Feel free to look at [notebook 04a](#) and [notebook 05b](#) for augmentation examples. There is also documentation for the [TorchVision Transforms class](#).

Hint: Remember not to make the data augmentation too extreme.

```
In [105... IMG_WIDTH, IMG_HEIGHT = (224, 224)  
  
# random_trans = transforms.Compose([  
#     transforms.RandomRotation(25),  
#     transforms.RandomResizedCrop((IMG_WIDTH, IMG_HEIGHT), scale=(.8, 1), ratio=(1, 1)),  
#     transforms.RandomHorizontalFlip(),  
#     transforms.ColorJitter(brightness=.2, contrast=.2, saturation=.2, hue=.2)])  
  
IMG_WIDTH, IMG_HEIGHT = (224, 224)
```

```

random_trans = transforms.Compose([
    transforms.RandomResizedCrop(IMG_WIDTH), # Random resize and crop
    transforms.RandomHorizontalFlip(),       # Random horizontal flip
    transforms.ToTensor(),                   # Convert the image to a tensor
    transforms.Normalize(                   # Normalize the image to the ImageNet mean and std
        mean=[0.485, 0.456, 0.406],         # Mean for ImageNet
        std=[0.229, 0.224, 0.225]          # Standard deviation for ImageNet
    )
])

```

7.7 Load Dataset

Now it's time to load the train and validation datasets.

In [106... DATA_LABELS = ["freshapples", "freshbanana", "freshoranges", "rottenapples", "rottenbanana", "rottenoranges"]

```

class MyDataset(Dataset):
    def __init__(self, data_dir):
        self.imgs = []
        self.labels = []

        for l_idx, label in enumerate(DATA_LABELS):
            data_paths = glob.glob(data_dir + label + '/*.png', recursive=True)
            for path in data_paths:
                img = tv_io.read_image(path, tv_io.ImageReadMode.RGB)
                self.imgs.append(pre_trans(img).to(device))
                self.labels.append(torch.tensor(l_idx).to(device))

    def __getitem__(self, idx):
        img = self.imgs[idx]
        label = self.labels[idx]
        return img, label

    def __len__(self):
        return len(self.imgs)

```

Select the batch size `n` and set `shuffle` either to `True` or `False` depending on if we are training or validating. For a reference, check out [notebook 05b](#).

In [107...

```
n = 32

train_path = "data/fruits/train/"
train_data = MyDataset(train_path)
train_loader = DataLoader(train_data, batch_size=n, shuffle=True)
train_N = len(train_loader.dataset)

valid_path = "data/fruits/valid/"
valid_data = MyDataset(valid_path)
valid_loader = DataLoader(valid_data, batch_size=n, shuffle=False)
valid_N = len(valid_loader.dataset)
```

7.8 Train the Model

Time to train the model! We've moved the `train` and `validate` functions to our [utils.py](#) file. Before running the below, make sure all your variables are correctly defined.

It may help to rerun this cell or change the number of `epochs`.

In [108...

```
epochs = 20

for epoch in range(epochs):
    print('Epoch: {}'.format(epoch))
    utils.train(my_model, train_loader, train_N, random_trans, optimizer, loss_function)
    utils.validate(my_model, valid_loader, valid_N, loss_function)
```

```
Epoch: 0
Train - Loss: 9.0149 Accuracy: 0.9289
Valid - Loss: 2.9771 Accuracy: 0.9058
Epoch: 1
Train - Loss: 8.4396 Accuracy: 0.9323
Valid - Loss: 2.7823 Accuracy: 0.9088
Epoch: 2
Train - Loss: 7.7615 Accuracy: 0.9399
Valid - Loss: 3.4755 Accuracy: 0.8967
Epoch: 3
Train - Loss: 9.2284 Accuracy: 0.9306
Valid - Loss: 3.5803 Accuracy: 0.8997
Epoch: 4
Train - Loss: 8.3608 Accuracy: 0.9264
Valid - Loss: 3.7249 Accuracy: 0.8723
Epoch: 5
Train - Loss: 8.5347 Accuracy: 0.9272
Valid - Loss: 2.8393 Accuracy: 0.9058
Epoch: 6
Train - Loss: 10.6540 Accuracy: 0.9171
Valid - Loss: 3.1791 Accuracy: 0.9058
Epoch: 7
Train - Loss: 7.7732 Accuracy: 0.9289
Valid - Loss: 2.7630 Accuracy: 0.9179
Epoch: 8
Train - Loss: 7.7500 Accuracy: 0.9281
Valid - Loss: 2.4242 Accuracy: 0.9301
Epoch: 9
Train - Loss: 7.1289 Accuracy: 0.9365
Valid - Loss: 3.5560 Accuracy: 0.9058
Epoch: 10
Train - Loss: 7.6141 Accuracy: 0.9340
Valid - Loss: 3.3394 Accuracy: 0.9088
Epoch: 11
Train - Loss: 7.7173 Accuracy: 0.9357
Valid - Loss: 3.2930 Accuracy: 0.9119
Epoch: 12
Train - Loss: 7.7512 Accuracy: 0.9349
Valid - Loss: 3.6795 Accuracy: 0.9058
Epoch: 13
Train - Loss: 7.2633 Accuracy: 0.9315
```



```
Valid - Loss: 3.1034 Accuracy: 0.9271
Epoch: 14
Train - Loss: 9.8137 Accuracy: 0.9272
Valid - Loss: 4.6080 Accuracy: 0.8875
Epoch: 15
Train - Loss: 5.7352 Accuracy: 0.9552
Valid - Loss: 2.7930 Accuracy: 0.9271
Epoch: 16
Train - Loss: 7.5913 Accuracy: 0.9323
Valid - Loss: 2.8340 Accuracy: 0.9331
Epoch: 17
Train - Loss: 6.7553 Accuracy: 0.9442
Valid - Loss: 3.6448 Accuracy: 0.8997
Epoch: 18
Train - Loss: 5.4216 Accuracy: 0.9535
Valid - Loss: 2.4536 Accuracy: 0.9362
Epoch: 19
Train - Loss: 6.4230 Accuracy: 0.9475
Valid - Loss: 3.8292 Accuracy: 0.8997
```

7.9 Unfreeze Model for Fine Tuning

If you have reached 92% validation accuracy already, this next step is optional. If not, we suggest fine tuning the model with a very low learning rate.

```
In [112... # Unfreeze the base model
vgg_model.requires_grad_(True)
optimizer = Adam(my_model.parameters(), lr=.0001)
```

```
In [113... epochs = 5

for epoch in range(epochs):
    print('Epoch: {}'.format(epoch))
    utils.train(my_model, train_loader, train_N, random_trans, optimizer, loss_function)
    utils.validate(my_model, valid_loader, valid_N, loss_function)
```

```
Epoch: 0
Train - Loss: 5.7586 Accuracy: 0.9459
Valid - Loss: 3.2667 Accuracy: 0.9210
Epoch: 1
Train - Loss: 4.3099 Accuracy: 0.9577
Valid - Loss: 2.9860 Accuracy: 0.9210
Epoch: 2
Train - Loss: 6.6279 Accuracy: 0.9399
Valid - Loss: 2.9355 Accuracy: 0.9240
Epoch: 3
Train - Loss: 6.5282 Accuracy: 0.9433
Valid - Loss: 2.8818 Accuracy: 0.9301
Epoch: 4
Train - Loss: 7.0402 Accuracy: 0.9315
Valid - Loss: 3.0086 Accuracy: 0.9210
```

7.10 Evaluate the Model

Hopefully, you now have a model that has a validation accuracy of 92% or higher. If not, you may want to go back and either run more epochs of training, or adjust your data augmentation.

Once you are satisfied with the validation accuracy, evaluate the model by executing the following cell. The evaluate function will return a tuple, where the first value is your loss, and the second value is your accuracy. To pass, the model will need have an accuracy value of 92% or higher .

```
In [114... utils.validate(my_model, valid_loader, valid_N, loss_function)
```

```
Valid - Loss: 3.0086 Accuracy: 0.9210
```

7.11 Run the Assessment

To assess your model run the following two cells.

NOTE: `run_assessment` assumes your model is named `my_model` . If for any reason you have modified these variable names, please update the names of the arguments passed to `run_assessment` .

```
In [115... from run_assessment import run_assessment
```

```
In [116... run_assessment(my_model)
```

Evaluating model to obtain average accuracy...


Accuracy: 0.9210

Accuracy required to pass the assessment is 0.92 or greater.
Your average accuracy is 0.9210.

Congratulations! You passed the assessment!
See instructions below to generate a certificate.

7.12 Generate a Certificate

If you passed the assessment, please return to the course page (shown below) and click the "ASSESS TASK" button, which will generate your certificate for the course.

No description has been provided for this image

Header