Cassava Leaf Disease Classification

Introduction

Cassava is the second-largest provider of carbohydrates in Africa, it is a key food security crop grown by smallholder farmers because it can withstand harsh conditions. At least 80% of household farms in Sub-Saharan Africa grow this starchy root, but viral diseases are major sources of poor yields. The dataset was acquired from Kaggle and is part of a competition in order to detect which disease the Cassava leaves have. The project was written in a jupyter notebook using the python programming language, with additional machine learning libraries and wrappers such as pytorch, torchvision, pandas, matplotlib, tez, and others.

The scope and objective of the task is to be able to classify in which category a picture of a Cassava leaf is part of, five classes being in total. This turns out to be a multi-class classification problem and we will solve it using a ResNet18 model.

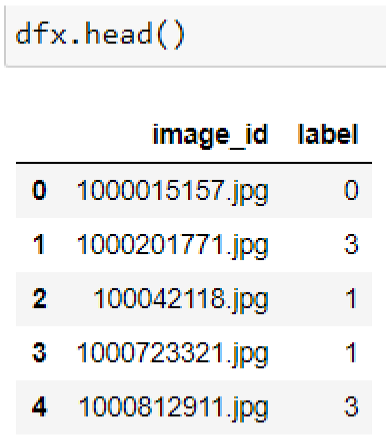
The Dataset

The dataset contains a total of 21,397 images with specific labels for each image. There are 5 different classes in the dataset that are available for this classification problem:

- Class 0 represents the "Cassava Bacterial Blight" and has 1087 related images
- Class 1 represents the "Cassava Brown Streak Disease" and has 2189 related images
- Class 2 represents the "Cassava Green Mottle" and has 2386 related images
- Class 3 represents the "Cassava Mosaic Disease" and has 13158 related images
- Class 4 represents the "Healthy" and has 2577 related images

An observation could be drawn that class 3 is biased and contains a number of images of more than 12 times greater than the class 0 one.

We can view some data from the training.csv file as a table that looks as follows in the below image:



```
dfx.head()
```

	image_id	label
0	1000015157.jpg	0
1	1000201771.jpg	3
2	100042118.jpg	1
3	1000723321.jpg	1
4	1000812911.jpg	3

Head of the data from the train.csv file

Taking a look at an image from the input data, it looks like this:

```
plot_image(train_dataset[0])
```

```
tensor(1)
```

Show the image on index 0 from the input data

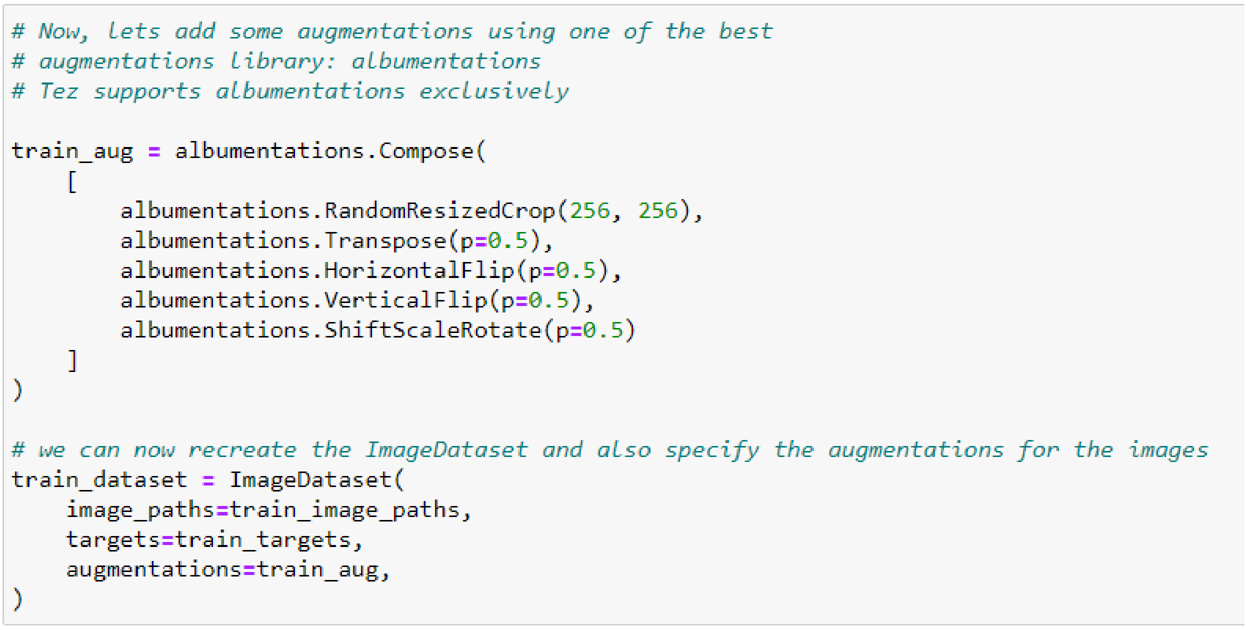
Since there are a lot of images that do not have complete keypoints annotations we can iterate through some of the to see how they appear on screen:

Data augmentations

In order to improve the accuracy of the model, we can use a technique that allows us to get more data from the dataset that we already have. For this we can use augmentations - this technique is used in order to modify images so the model can be trained better (for example, cropped images, horizontally flipped images, vertically flipped images, shifted images, transposed images, etc.)

For this purpose we can use a great library that is able to augmentate the images for us easily - it is called albumentations. Albumentations can also be used by the tez library to create the ImageDataset.

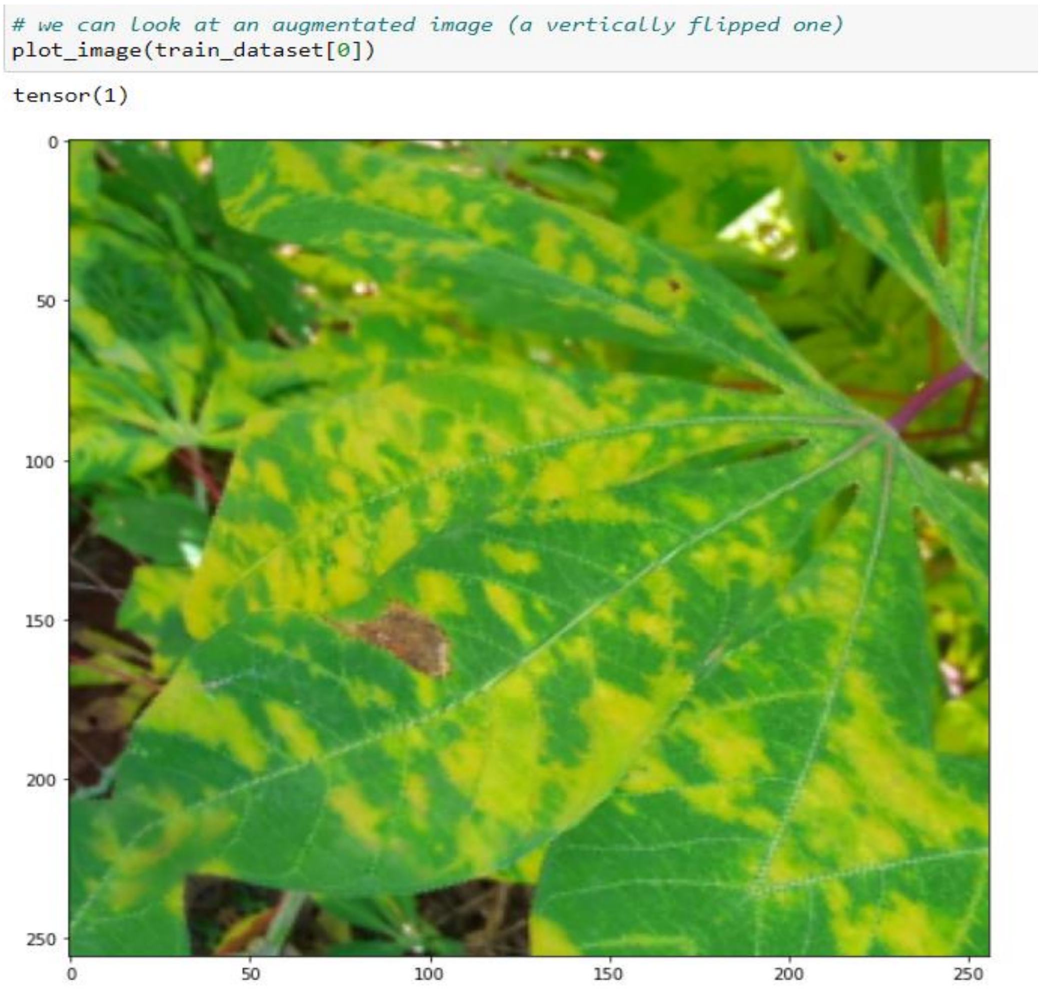
The data augmentations for this problem were created as in the picture below:



```python
# Now, lets add some augmentations using one of the best
# augmentations library: albumentations
# Tez supports albumentations exclusively

train_aug = albumentations.Compose(
    [
        albumentations.RandomResizedCrop(256, 256),
        albumentations.Transpose(p=0.5),
        albumentations.HorizontalFlip(p=0.5),
        albumentations.VerticalFlip(p=0.5),
        albumentations.ShiftScaleRotate(p=0.5)
    ]
)

# we can now recreate the ImageDataset and also specify the augmentations for the images
train_dataset = ImageDataset(
    image_paths=train_image_paths,
    targets=train_targets,
    augmentations=train_aug,
)
```

Augmentations for the training data composed of multiple instances of augmentations

Now we can take a look at an example of augmentation:



```
# we can look at an augmentated image (a vertically flipped one)
plot_image(train_dataset[0])
```

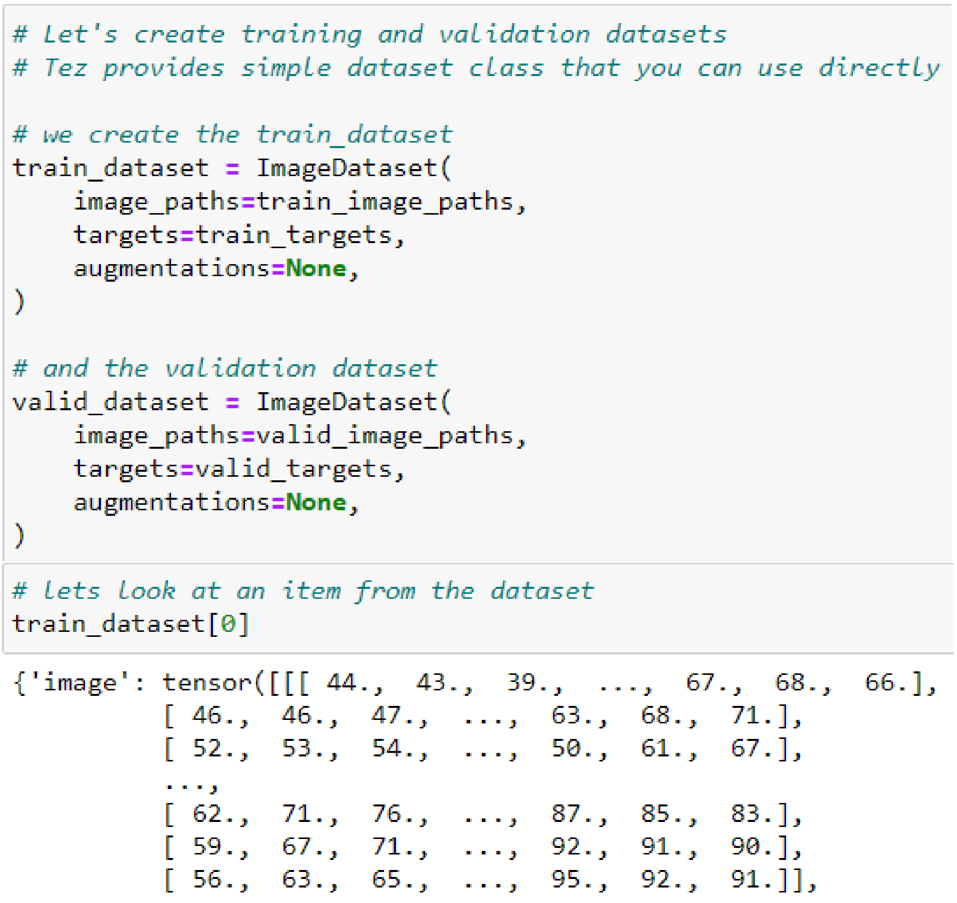
```
tensor(1)
```

Vertical flip augmentation example

Data processing

The data processing will be done using pytorch datasets, not directly, but by using the tez library which provides a helpful class of ImageDataset, specific for these type of problems which is quite helpful.

The data from the csv file and the images themselves need to be converted to tensors for the dataset:



```python
# Let's create training and validation datasets
# Tez provides simple dataset class that you can use directly

# we create the train_dataset
train_dataset = ImageDataset(
    image_paths=train_image_paths,
    targets=train_targets,
    augmentations=None,
)

# and the validation dataset
valid_dataset = ImageDataset(
    image_paths=valid_image_paths,
    targets=valid_targets,
    augmentations=None,
)
```

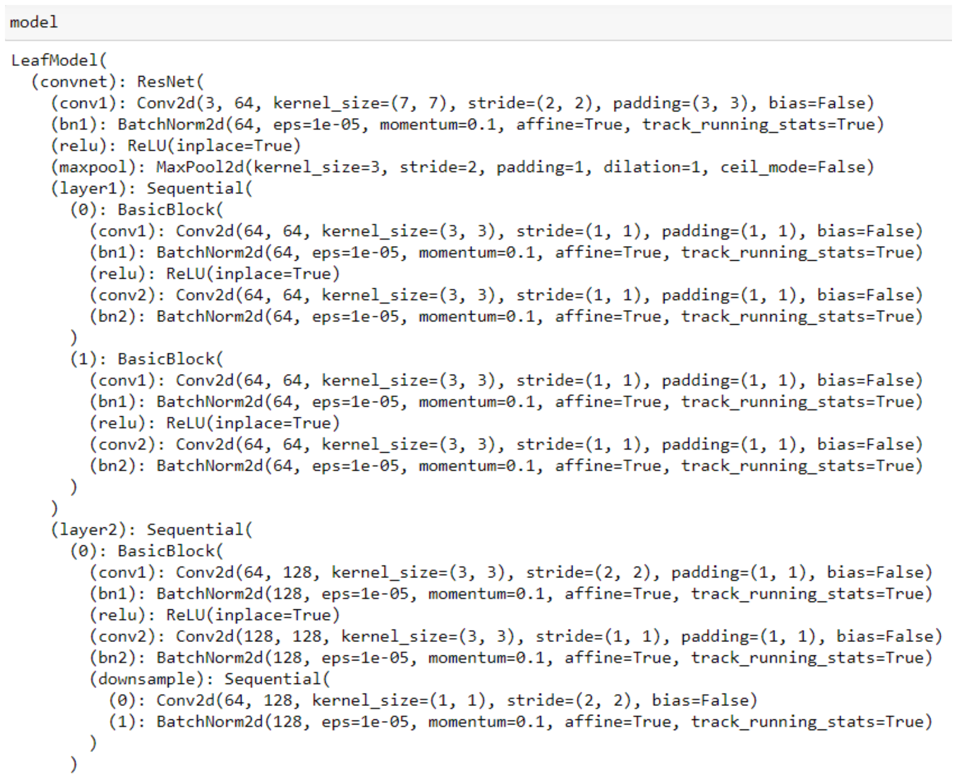
```python
# lets look at an item from the dataset
train_dataset[0]
```

```
{'image': tensor([[[ 44.,  43.,  39.,  ...,  67.,  68.,  66.],
          [ 46.,  46.,  47.,  ...,  63.,  68.,  71.],
          [ 52.,  53.,  54.,  ...,  50.,  61.,  67.],
          ...,
          [ 62.,  71.,  76.,  ...,  87.,  85.,  83.],
          [ 59.,  67.,  71.,  ...,  92.,  91.,  90.],
          [ 56.,  63.,  65.,  ...,  95.,  92.,  91.]],
```

ImageDataset example of tensors from index 0

The Model

The model is based on ResNet18 and a part of the structure can be observed in the picture below. The resnet18 was taken from torchvision.models.resnet18 and is a pretrained model, the fully connected layer of the network is replaced by a linear layer which has as output 5 possible values.



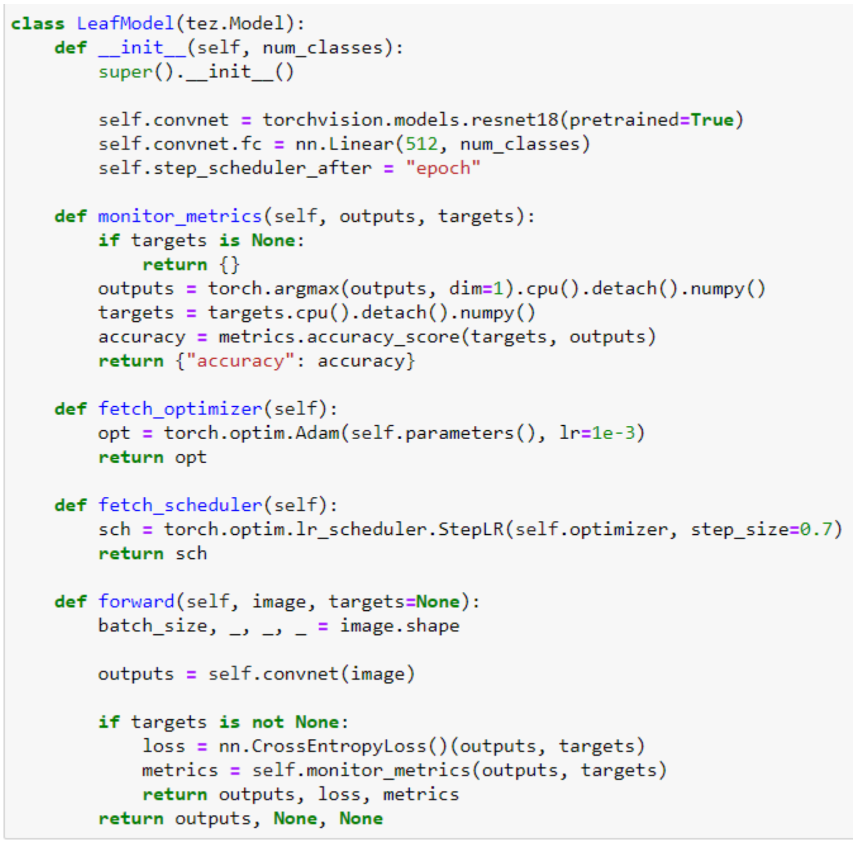
```
model
```

```
LeafModel(
  (convnet): ResNet(
    (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
    (layer1): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (layer2): Sequential(
      (0): BasicBlock(
        (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (downsample): Sequential(
          (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
          (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        )
      )
```

ResNet18 structure from torchvision.models

The implementation of the LeafModel which uses the resnet18 network is created in such way to accommodate the inputs for this problem, in this case the images and targets or classes for the images, given as parameters.

The class can be observed in the picture below, which contains the implementation of the LeafModel class:

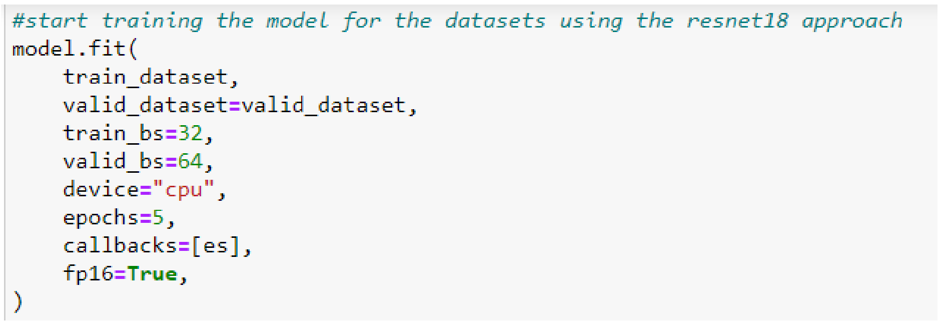


```python
class LeafModel(tez.Model):
    def __init__(self, num_classes):
        super().__init__()

        self.convnet = torchvision.models.resnet18(pretrained=True)
        self.convnet.fc = nn.Linear(512, num_classes)
        self.step_scheduler_after = "epoch"

    def monitor_metrics(self, outputs, targets):
        if targets is None:
            return {}
        outputs = torch.argmax(outputs, dim=1).cpu().detach().numpy()
        targets = targets.cpu().detach().numpy()
        accuracy = metrics.accuracy_score(targets, outputs)
        return {"accuracy": accuracy}

    def fetch_optimizer(self):
        opt = torch.optim.Adam(self.parameters(), lr=1e-3)
        return opt

    def fetch_scheduler(self):
        sch = torch.optim.lr_scheduler.StepLR(self.optimizer, step_size=0.7)
        return sch

    def forward(self, image, targets=None):
        batch_size, _, _, _ = image.shape

        outputs = self.convnet(image)

        if targets is not None:
            loss = nn.CrossEntropyLoss()(outputs, targets)
            metrics = self.monitor_metrics(outputs, targets)
            return outputs, loss, metrics
        return outputs, None, None
```

Implementation of the LeafModel class

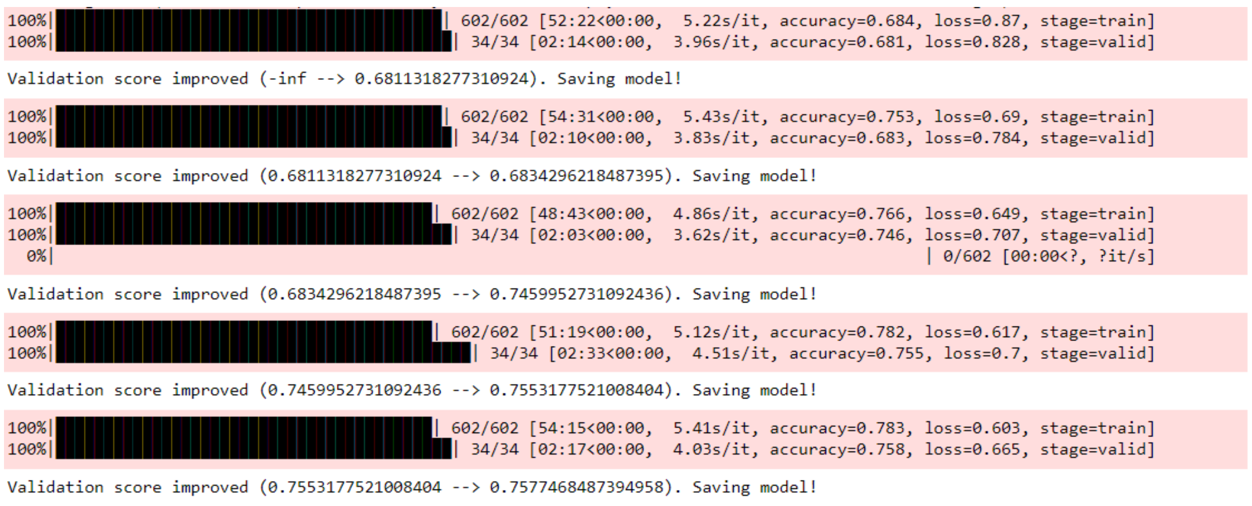
The results

After the training has finished for the implemented LeafModel using the pretrained ResNet18 network, we can observe the promising results of the model, which has only been trained for 5 epochs:



```python
#start training the model for the datasets using the resnet18 approach
model.fit(
    train_dataset,
    valid_dataset=valid_dataset,
    train_bs=32,
    valid_bs=64,
    device="cpu",
    epochs=5,
    callbacks=[es],
    fp16=True,
)
```

Starting the model training



```
100%|████████████████████████████████████████| 602/602 [52:22<00:00,  5.22s/it, accuracy=0.684, loss=0.87, stage=train]
100%|█████████████████████████████████████████| 34/34 [02:14<00:00,  3.96s/it, accuracy=0.681, loss=0.828, stage=valid]

Validation score improved (-inf --> 0.6811318277310924). Saving model!

100%|████████████████████████████████████████| 602/602 [54:31<00:00,  5.43s/it, accuracy=0.753, loss=0.69, stage=train]
100%|█████████████████████████████████████████| 34/34 [02:10<00:00,  3.83s/it, accuracy=0.683, loss=0.784, stage=valid]

Validation score improved (0.6811318277310924 --> 0.6834296218487395). Saving model!

100%|███████████████████████████████████████| 602/602 [48:43<00:00,  4.86s/it, accuracy=0.766, loss=0.649, stage=train]
100%|█████████████████████████████████████████| 34/34 [02:03<00:00,  3.62s/it, accuracy=0.746, loss=0.707, stage=valid]
  0%|                                                                                          | 0/602 [00:00<?, ?it/s]

Validation score improved (0.6834296218487395 --> 0.7459952731092436). Saving model!

100%|███████████████████████████████████████| 602/602 [51:19<00:00,  5.12s/it, accuracy=0.782, loss=0.617, stage=train]
100%|███████████████████████████████████████████| 34/34 [02:33<00:00,  4.51s/it, accuracy=0.755, loss=0.7, stage=valid]

Validation score improved (0.7459952731092436 --> 0.7553177521008404). Saving model!

100%|███████████████████████████████████████| 602/602 [54:15<00:00,  5.41s/it, accuracy=0.783, loss=0.603, stage=train]
100%|█████████████████████████████████████████| 34/34 [02:17<00:00,  4.03s/it, accuracy=0.758, loss=0.665, stage=valid]

Validation score improved (0.7553177521008404 --> 0.7577468487394958). Saving model!
```

The results of the 5 epochs of training the model

As can be seen in the picture above, the model shows promising results for as little as 5 epochs of training and the loss is getting lower with each epoch passing, the same as the accuracy is increasing with each epoch slightly.