Udacity Machine Learning Nanodegree 2020

Capstone Project:

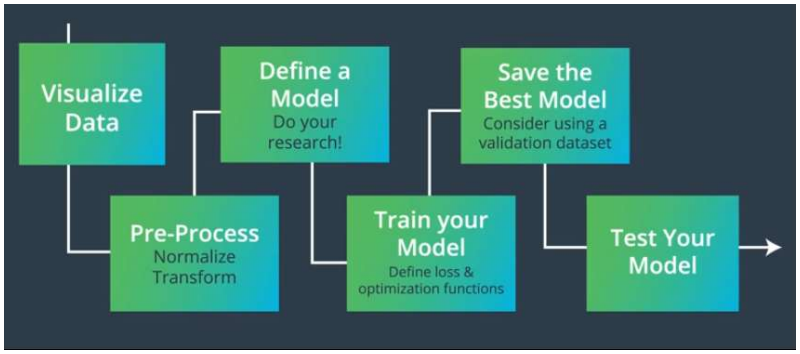
Plant Pathology Identification

Author: Angel Peinado Perez, May 2020.

1. INTRODUCITION

This project is based on Kaggle competition "Plant Pathology 2020", for identification of the category of foliar diseases in apple trees (https://www.kaggle.com/c/plant-pathology-2020-fgvc7/overview). This competition is a challenge consisting to distinguish between leaves which are healthy, those which are infected with apple rust, those that have apple scab, and those with more than one disease.

Project can be follow in a Jupiter notebook presented in the complete project documentation (leaf_project.ipynb), which follows the following project solution structure:



1.1 Motivation

After implementing successfully "Dog breed project" available in Udacity platform, I preferred to present an original project as Capstone Project with a more challenging approach in order to apply part of the knowledge acquired during the nanodegree. In the case of the present project, selection of a multi label classification problem was a personal challenge.

Checking multiple competitions in Kaggle, I decided this project about plants and image recognition, which both are topics I am very interested about (both are hobbies). Also, it was important for the selection, the relatively low reduced size of the dataset (800 Mb aprox) which is possible to manage with GPU open cloud services available in the internet (Google Colab).

1.2 Problem Statement

Objectives of 'Plant Pathology Challenge' are to train a model using images of training dataset to:

- Accurately classify a given image from testing dataset into different diseased category or a healthy leaf;
- Accurately distinguish between many diseases, sometimes more than one on a single leaf;
- Deal with rare classes and novel symptoms;
- Address depth perception—angle, light, shade, physiological age of the leaf;
- Incorporate expert knowledge in identification, annotation, quantification, and guiding computer vision to search for relevant features during learning.

Misdiagnosis of the many diseases impacting agricultural crops can lead to misuse of chemicals leading to the emergence of resistant pathogen strains, increased input costs, and more outbreaks with significant economic loss and environmental impacts.

Current disease diagnosis based on human scouting is time-consuming and expensive, and although computer-vision based models have the promise to increase efficiency, the great variance in symptoms due to age of infected tissues, genetic variations, and light conditions within trees decreases the accuracy of detection.

1.3 Project proposal

All the pre-requisites in the proposal presented were accomplished.

1.4 Metrics

The evaluation metrics to be used will be mainly Accuracy Score: Number of correct predictions / Total number of predictions.

2. ANALYSIS

2.1 Data Exploration

2.1.1 Input data

Input data used for his project was extracted from Kaggle competition. Files available are images and csv with only 780 Mb size.

- train.csv (CSV file with information of training images and columns with information about target diseases)

	image_id	healthy	multiple_diseases	rust	scab
0	Train_0	0	0	0	1
1	Train_1	0	1	0	0
2	Train_2	1	0	0	0
3	Train_3	0	0	1	0
4	Train_4	1	0	0	0

  - image_id: name of image files for train

- combination of multiple diseases: 1 label if True
- healthy: 1 label if True
- rust: 1 label if True
- scab: 1 label if True

➢ Images

Folder containing the 1820 train and 1820 test images (2048 x 1365 pix), in jpg format.

➢ Test.csv

- image_id: name of image files for testing, without labels information because these are the object of prediction.

	image_id
0	Test_0
1	Test_1
2	Test_2
3	Test_3
4	Test_4

➢ sample_submission.csv (Object submission file example)

- image_id: name of image files for testing
- combination of multiple diseases: probability of multiple diseases
- healthy: probability of healthy
- rust: probability of rust
- scab: probability of scab

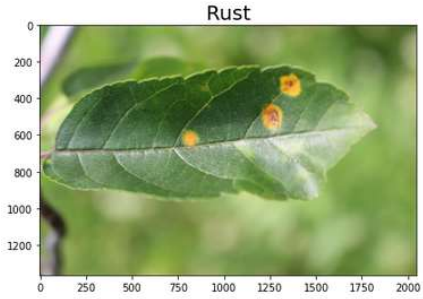
	image_id	healthy	multiple_diseases	rust	scab
0	Test_0	0.25	0.25	0.25	0.25
1	Test_1	0.25	0.25	0.25	0.25
2	Test_2	0.25	0.25	0.25	0.25
3	Test_3	0.25	0.25	0.25	0.25
4	Test_4	0.25	0.25	0.25	0.25

Note: it is asked to submit probabilities of disease

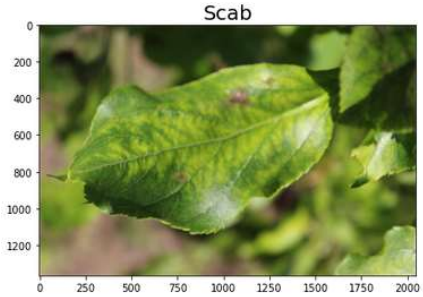
2.1.2 Explanation of Diseases

The explanation of the diseases object of the present project are the following:

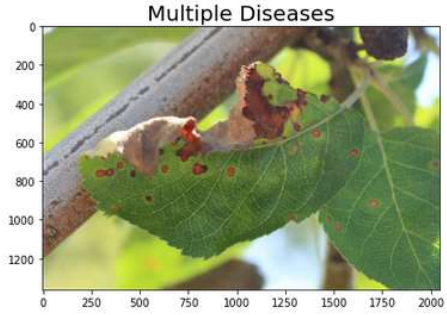
Rust: Common rust (*Phragmidium* spp.) is a fungal disease that attacks roses, hollyhocks, snapdragons, daylilies, beans, tomatoes and lawns. It is most often found on mature plants where symptoms appear primarily on the surfaces of lower leaves.



Scab: Caused by a fungus that infects both leaves and fruit. Scabby fruit are often unfit for eating. Infected leaves have olive green to brown spots. Leaves with many leaf spots turn yellow and fall off early.



Multiple disease: combinations of Scab and rust and/or other diseases



2.2 Exploratory Visualization

2.2.1 Label Encoder

A label encoder will be used to convert a multi-label classification problem to a single label classification (with the creation of "Target" column).

Sklearn preprocessing tool “LabelEncoder” will be used to create a Target column with one-hot encoding (Encode target labels with value between 0 and n_classes-1)

```
df['Target'] = target
le = LabelEncoder()
df['Target'] = le.fit_transform(df['Target'])
df.head()
```

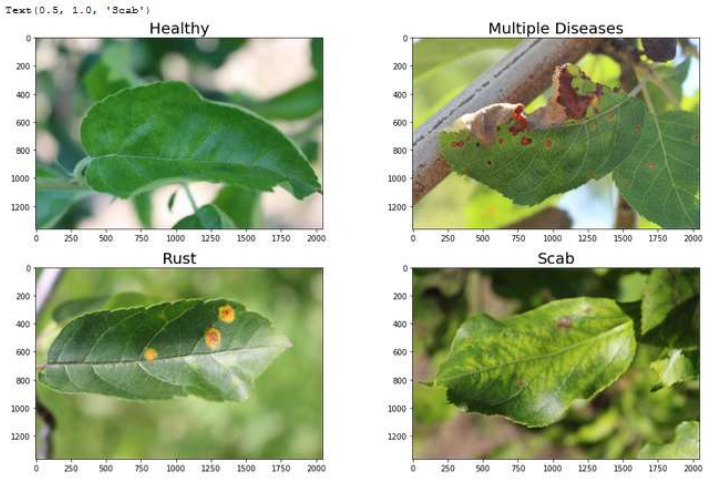
	image_id	healthy	multiple_diseases	rust	scab	Target
0	Train_0	0	0	0	1	3
1	Train_1	0	1	0	0	1
2	Train_2	1	0	0	0	0
3	Train_3	0	0	1	0	2
4	Train_4	1	0	0	0	0

2.2.2 Visualization of images

First, it is important to take a look of the images object of work in order to see if they are properly loaded and the homogeneity:

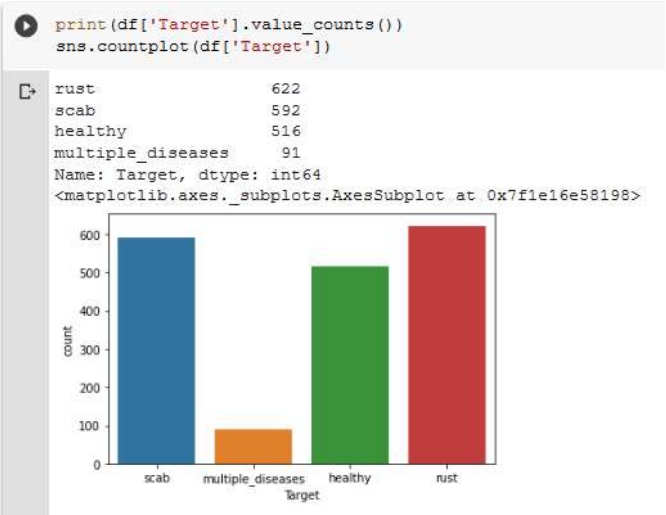
```
images = glob.glob('./images/*.jpg')
for i in range(5):
    image = cv2.imread(images[i])
    plt.figure(figsize=(12,5))
    plt.subplot(2,4,i+1)
    plt.xticks([])
    plt.yticks([])
    plt.title(images[i])
    plt.imshow(image)
```

An also visualization of the images with the labels of the diseases:



2.2.3 Exploration of data

Using a bar plot is possible explore the frequency of each disease in the train dataset of images:



As it can be seen the data is balanced between the different types of plant health states and diseases.

2.3 Algorithms and Techniques

The proposed solution to this problem is the application of Deep Learning techniques that have been proved to be highly successful in the field of image classification.

Classifier that will be used for this project is Convolutional Neural Networks (CNN) built with Pytorch. CNNs are the class of deep learning techniques most commonly used to analyze visual imagery.

A CNN architecture has typically: convolutional layers, ReLU layers, Pooling layers, Fully connected layers and Dropout layers. Number of convolutional layers and CNN parameters can be selected to train the model and obtain the best metrics (results) of the classifier.

Typical parameters to be selected for training are:

- Training length (Number of epochs in training)
- Batch size: quantity of data (images or CSV lines) to be used in a training step.
- Learning rate: how much to change the model in response to the estimated error each time the model weights are updated
- Dropout rate: regularization method to avoid overfitting. It is the probability of node deactivation.

First a model from scratch will be build, trained and tested to check the accuracy, and later transfer learning techniques will be used to improve the accuracy.

This project will be developed using Google Colab services, with the GPU available.

2.4 Benchmark

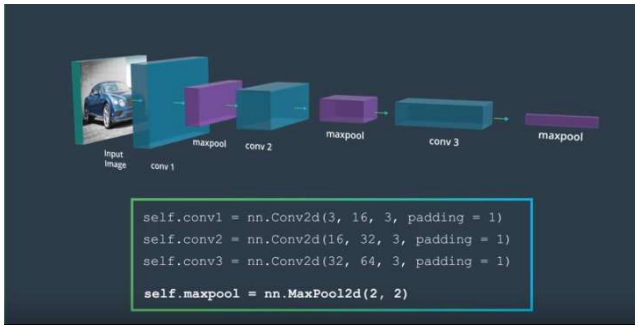
2.4.1 Model from Scratch

First a model from scratch will be build using convolutional layers, ReLU layers, Pooling layers, Fully connected layers and Dropout layers. Object of this section is putting in practice the content of the Nanodregree regarding CNN architecture.

Then this model will be trained and tested with the validation data to check the accuracy (metrics used for evaluation). As it will be seen, it is not expected to obtain an acceptable result a later on transfer learning techniques will be needed to improve the metrics.

Regarding CNN architecture, in the forward function can be seen the following:

- Input is the image converted to tensor 224x224x3 (after application of tranforms in the dataset, as it will be seen in section 3.1.3)
- 4 Convolutional layers with 3 kernel (3x3 most commonly used), and padding = 1. Padding = 1 will produce the same dimension of image after the convolutional layer (pixels added to an image when it is being processed by the kernel)
- ReLU layers applied for every convolutional
- Maxpooling layers 2x2, to reduce the spatial dimension of the input volume for next layers



- 2 linear layers that transforms the output of the convolutional/maxpooling layers, first layer from 25088 (128 * 14 * 14) to 5000 and second layer from 5000 to 4, because there are 4 different possible outputs resulting the Label Encoder (Target 0, 1, 2 and 3)
- Also, there is a dropout layer after the first linear layer for avoiding overfitting.

Architecture in notebook can be seen here:

```
# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        # convolutional layer (sees the inicial image converted to 224x224x3 image tensor)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        # convolutional layer (sees 112x112x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        # convolutional layer (sees 56x56x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        # convolutional layer (sees 28x28x32 tensor)
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (128 * 14 * 14 -> 25088)
        self.fc1 = nn.Linear(25088, 5000)
        # linear layer (5000 -> 4 types of 'Targets' (0,1,2,3))
        self.fc2 = nn.Linear(5000, 4)
        # dropout layer (p=0.2)
        self.dropout = nn.Dropout(0.2)
```

```
    def forward(self, x):
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        # flatten image input
        x = x.view(-1, 128 * 14 * 14)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)
        return x

# instantiate the CNN
model_scratch = Net()
```

2.4.2 Models using transfer learning

The second approach is the using of transfer learning technics to take advantage of the models available in Pytorch that were pretrained with images datasets (i.e ImageNet, etc)

A pre-trained model has been previously trained on a dataset and contains the weights and biases that represent the features of whichever dataset it was trained on. Learned features are often transferable to different data. For example, a model trained on a large dataset of bird images will contain learned features like edges or horizontal lines that you would be transferable to our dataset of Plant images.

After reading multiples referenced articles about which models were pertained with similar data and evaluate which of them could provide best results, Densenet121 and Resnet 18 were selected.

2.4.2.1 Densenet 121

DenseNets (Dense Convolutional Network) have several compelling advantages: they alleviate the vanishing-gradient problem, strengthen feature propagation, encourage feature reuse, and substantially reduce the number of parameters. Densenet has weights pre-trained on ImageNet and 121 is the depth of layers.

In the case of Densenet, it is needed to define the classifier with an output of 4 (number of targets in our problem)

```
model_transfer_1 = models.densenet121(pretrained=True)

for param in  model_transfer_1.parameters():
    param.require_grad = False

# Get the input of the last layer of Densenet121
n_inputs = model_transfer_1.classifier.in_features

# Create a new layer(n_inputs -> 4)
# The new layer's requires_grad will be automatically True.
last_layer = nn.Linear(n_inputs, 4)

# Change the last layer to the new layer.
model_transfer_1.classifier = last_layer
```

2.4.2.2 Resnet 18

ResNet (Residual deep neural networks) uses skip connections, and shortcuts to jump over some layers. Typical ResNet models are implemented with double- or triple- layer skips that contain nonlinearities (ReLU) and batch normalization in between ResNet-18 is a convolutional neural network that is 18 layers deep.

ResNet was pretrained version of the network trained on more than a million images from the ImageNet database.

In the case of Resnet, instead of classifier, it is needed to define the output of the final dense layer as 4 (number of targets in our problem).

```
## TODO: Specify model architecture
model_transfer_2 = models.resnet18(pretrained=True)

for param in  model_transfer_2.parameters():
    param.require_grad = False

# Create a new layer(n_inputs -> 4)
# The new layer's requires_grad will be automatically True.
model_transfer_2.fc.out_features=4

# Print the model.
print(model_transfer_2)
```

3. METHODOLOGY

3.1 Data preprocessing

3.1.1 Datasets creation

Creation of datasets is done with the following "Class objects" for Training/validation and test respectively:

- Training/Validation datasets:

```
class PlantDataset(Dataset):
    def __init__(self,csv,transformer):
        self.data = csv
        self.transform = transformer

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

    def __getitem__(self,idx):
        image = Image.open('./images/'+self.data.loc[idx]['image_id']+'.jpg')
        image = self.transform(image)
        labels = self.data['Target'].iloc[idx]
        return {'images':image,'labels':labels}
```

- Test dataset:

```
class PlantDataset_test(Dataset):
    def __init__(self,csv,transformer):
        self.data = csv
        self.transform = transformer

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

    def __getitem__(self,idx):
        image = Image.open('./images/'+self.data.loc[idx]['image_id']+'.jpg')
        image = self.transform(image)
        return {'images':image}
```

3.1.2 Batch size

Size of the batch to be managed by datasets in dataloaders is also defined as 20.

3.1.3 Data augmentation

Data augmentation is defined with the following Torchvision "transforms" for training, validation and testing:

```
train_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])])

test_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

valid_transforms = transforms.Compose([transforms.Resize(255),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])
```

- Train data transform: Normalization, Random rotation of images 30º, crop image to 224x224 using center as reference and Random horizontal flip.

- Validation and test transform: Normalization, Resize to 255x255, crop to 224x224 using center as reference.

3.1.4 Dataloaders

Training and Validation samples are randomly generated. Validation sample is as default a 10% of the original training set.

```
[ ] #Funcion for separation of valid and training sample. default 10% for validation.
    def train_valid_split(perc_valid=0.10):
        indices = range(len(train_dataset))
        split = int(perc_valid*len(train_dataset))
        train_indices = indices[split:]
        valid_indices = indices[:split]
        return train_indices, valid_indices
```

```
[ ] train_indices, valid_indices = train_valid_split()
```

```
train_sampler = SubsetRandomSampler(train_indices)
valid_sampler = SubsetRandomSampler(valid_indices)
```

```
train_loader = DataLoader(train_dataset,batch_size,sampler=train_sampler)
valid_loader = DataLoader(valid_dataset,batch_size,sampler=valid_sampler)
```

Finally, a dictionary with the loaders is defined. This is the dictionary that will be used in the training and validation loops:

```
[ ]  test_loader=DataLoader(test_dataset,batch_size)
```

```
loaders = {'train': train_loader,
           'valid': valid_loader,
          'test': test_loader}
```

3.2 Implementation

For the implementation of the training and the validation two different functions are used that will be explained in the following paragraphs.

3.2.1 Training function

```
[ ] def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
        """returns trained model"""
        # initialize tracker for minimum validation loss
        valid_loss_min = np.Inf
        train_losses_list, test_losses_list = [], []
        for epoch in range(1, n_epochs+1):
            # initialize variables to monitor training and validation loss
            train_loss = 0.0
            valid_loss = 0.0
```

Training function allows define the following arguments:

- Number of epochs
- Validation loader (dictionary of loaders previously defined)
- Model (scratch or transfer learning: Densenet121 or Resnet18)
- Optimizer: method for weight backdrop propagation used is CrossEntropy Loss. This optimizer measures the performance of a classification model and optimize the weights to reduce losses.
- Criterion: in this project used the SGD, Stochastic Gradient Descent
- Use_cuda: Use or GPU or CPU
- Save path, for saving model parameters
- Initialize the training and valid losses

```
    ####################
    # train the model #
    ####################
    model.train()
    for batch_idx, d in enumerate(loaders['train']):
        data = d['images']
        target = d['labels']
        # move to GPU
        if use_cuda:
            data = d['images'].cuda()
            target = d['labels'].cuda()
        ## find the loss and update the model parameters accordingly
        ## record the average training loss, using something like
        ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target.long())
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update training loss
        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))
```

This is the part of the function where model is trained in every epoch, with the following parts:

- Loading of batches
- Clear of gradients
- Forward pass: calculation of predicted output
- Calculation of batch loss
- Backward pass: calculation of gradients of the loss
- Perform a single optimization step (with optimizer)
- Update training loss

```
    ######################
    # validate the model #
    ######################
    model.eval()
    for batch_idx, d in enumerate(loaders['valid']):
        data = d['images']
        target = d['labels']
        # move to GPU
        if use_cuda:
            data = d['images'].cuda()
            target = d['labels'].cuda()
        ## update the average validation loss
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target.long())
        #loss = criterion(output, target)
        # update average validation loss
        valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)
    train_losses_list.append(train_loss)
    test_losses_list.append(valid_loss)
```

Part of the function where validation loss is calculated for every epoch, with the following parts:

- Model in mode “eval”: turn off gradients computation
- Forward pass: calculation of predicted output
- Calculation of batch loss
- Calculation of training loss accumulated for the epoch
- Calculation of validation loss accumulated for the epoch

```
    # print training/validation statistics
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch,
        train_loss,
        valid_loss
        ))

    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).    Saving model...'.
              format(valid_loss_min, valid_loss))
        torch.save(model.state_dict(), save_path)
        valid_loss_min = valid_loss
# return trained model
return model, train_losses_list, test_losses_list
```

In the last part of the function training and validation losses are printed for the epoch and model parameters are only saved if validation loss was decreased in this epoch.

3.2.2 Test function

Test function performs the following actions:

- takes as argument the trained model
- loading of the validation dataset in batches
- Forward pass: calculation of predicted output
- Calculation of validation loss accumulated for the complete validation dataset
- Print the total accuracy of the trained model in the validation dataset (%)

```
def test(model):
    test_loss = 0.
    correct = 0.
    total = 0.
    model.eval()
    for batch_idx, d in enumerate(loaders['valid']):
        data = d['images']
        target = d['labels']
        # move to GPU
        if use_cuda:
            data = d['images'].cuda()
            target = d['labels'].cuda()
        ## update the average validation loss
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion_transfer(output, target.long())
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        #pred = output.data.max(1, keepdim=True)[1]
        _, pred = torch.max(output, 1)
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)
        #print(output)
        #print(torch.softmax(output,1))
        #print("target",target)
        #print("prediction", pred)

    print('Test Loss: {:.6f}\n'.format(test_loss))

    print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
            100. * correct / total, correct, total))
```

3.3 Refinement

Three models are trained as explained in section 2.4:

- Model from Scratch
- Densenet 121
- ResNet 18

Learning rate defined is 0.01 as default. Other learning rates have been tested without significant improvement in the metrics.

Number of epochs considered for training was 20, and it was considered enough after checking the training and validation losses vs epochs curves.

4. RESULTS

4.1 Model Evaluation and Validation

4.1.1 Model from Scratch

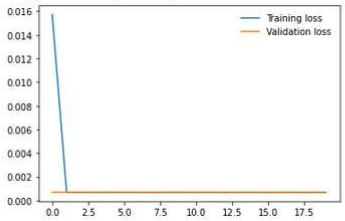
Results of the training and validation decrease dramatically in the first epoch and after that remains practically constant as it can be seen in the following image.

Training-Validation Error curve

Representation

```
train_losses_list = model_scratch[1]
test_losses_list = model_scratch[2]
plt.plot(train_losses_list, label='Training loss')
plt.plot(test_losses_list, label='Validation loss')
plt.legend(frameon=False)
```

```
<matplotlib.legend.Legend at 0x7f1e168246a0>
```



As it was expected the model performance is poor with an accuracy of 32% (59 hits of total validation set 182), only slight above the random prediction of 25% with a label target with 4 possibilities.

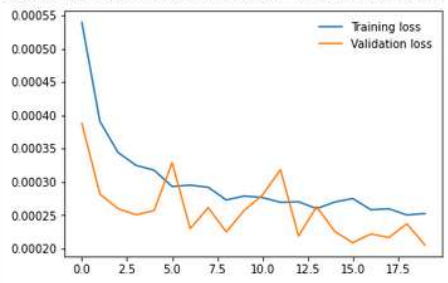
```
Test Loss: 1.260048


Test Accuracy: 32% (59/182)
```

4.1.2 Model 1 (Transfer learning: Densenet 121)

Densenet 121 performs very well and after 20 epochs training and validation loss curves are practically flat.

```
<matplotlib.legend.Legend at 0x7f1e10062b38>
```



Total accuracy of the model is 85% (155 hits of total validation set 182).

```
test(model=model_transfer_1)
```

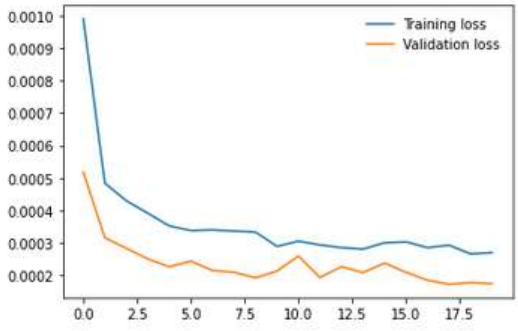
```
Test Loss: 0.382466


Test Accuracy: 85% (155/182)
```

4.1.3 Model 2 (Transfer learning: ResNet 18)

ResNet18 performs very even better and after 20 epochs training and validation loss curves are practically flat.

```
<matplotlib.legend.Legend at 0x7f1dfe47ccc0>
```



Total accuracy of the model is 85% (165 hits of total validation set 182).

```
[ ] test(model=model_transfer_2)
```

```
Test Loss: 0.340762

Test Accuracy: 90% (165/182)
```

4.2 Justification

Although model from scratch has a poor performance (32%), models using transfer learning achieve a very good results with a very high accuracy, close to 90%.

So, Model 1 and Model 2 (based on pretrained Densenet 121 and ResNet18) are acceptable for the purpose of the project.

4.3 Creation of submission file

Final part of the project (and the notebook) is the representation of the output according to the request of the Kaggle competition.

Competition request a CSV file with probabilities, so Softmax function shall be applied to the output.

Creating Submission File

Submission file will be created in the requested format (probabilities).

```
[ ] predict = []
    model_transfer_1.eval()
    for batch_idx, d in tqdm(enumerate(test_loader)):
        data = d['images'].cpu()
        model_transfer_1=model_transfer_1.cpu()
        output = model_transfer_1(data)
        #output = output.cpu().detach().numpy()
        #output = np.argmax(output)
        output=torch.softmax(output,1)
        predict.append(output)
        if batch_idx == 2:
          break
        #print(output)
```

```
[ ] healthy = []
    multiple_disease = []
    rust = []
    scab = []
    for i in tqdm(range(len(predict))):
        for j in range(batch_size):
            healthy.append(float(predict[i][j][0]))
            multiple_disease.append(float(predict[i][j][1]))
            rust.append(float(predict[i][j][2]))
            scab.append(float(predict[i][j][3]))
```

```
[ ] sample_submission = pd.read_csv('./sample_submission.csv')
    sample_submission.head()
```

	image_id	healthy	multiple_diseases	rust	scab
0	Test_0	0.25	0.25	0.25	0.25
1	Test_1	0.25	0.25	0.25	0.25
2	Test_2	0.25	0.25	0.25	0.25
3	Test_3	0.25	0.25	0.25	0.25
4	Test_4	0.25	0.25	0.25	0.25

```
sample_submission['healthy'][0:(len(healthy))]=healthy
sample_submission['multiple_diseases'][0:(len(healthy))] = multiple_disease
sample_submission['rust'][0:(len(healthy))] = rust
sample_submission['scab'][0:(len(healthy))] = scab
```

```
[ ] sample_submission.head(61)
```

	image_id	healthy	multiple_diseases	rust	scab
0	Test_0	0.019549	0.043784	0.011844	0.924823
1	Test_1	0.037881	0.103699	0.503928	0.354492
2	Test_2	0.379846	0.064521	0.510378	0.045255
3	Test_3	0.015751	0.010392	0.967870	0.005988
4	Test_4	0.710638	0.017337	0.258122	0.013903
...	...	...	...	...	...

```
[ ] sample_submission.to_csv('sample_submission.csv')
```

5. REFERENCES

[1.] The Plant Pathology 2020 challenge dataset to classify foliar disease of apples. Ranjita Thapa (1), Noah Snavely (2), Serge Belongie (2), Awais Khan (1) ((1) Plant Pathology and Plant-Microbe Biology Section, Cornell University, Geneva, NY, (2) Cornell Tech). https://arxiv.org/abs/2004.11958

[2.] https://www.planetnatural.com/pest-problem-solver/plant-disease/common-rust/

[3.] https://www.kaggle.com/c/plant-pathology-2020-fgvc7/overview

[4.] https://towardsdatascience.com/understanding-and-visualizing-densenets-7f688092391a

[5.] Deep Residual Learning for Image Recognition. Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. https://arxiv.org/abs/1512.03385