COMP SCI 7209 Big Data Analysis and Project

Image Analysis: Plant Seedlings Classification

Name	Shubham Gupta
Student No.	a1787223
Class	Ms. Data Science
Department	COMP SCI
Email	`shubham.gupta@student.adelaide.edu.au`
Date	October 28, 2020

THE UNIVERSITY *of* ADELAIDE

Abstract

After applying multiple models to the Plant seedling dataset, the best performing model was at accuracy of about 96%. The best performing model was a combination of ResNet50 model available via torchvision and custom made fully connected layer. I tried to implement other models as well on the dataset, but faced numerous errors. This project is done with PyTorch. I had no knowledge of Pytorch before this Assignment. Everything was learnt from scratch and a lot of references were taken from online community. I'm thankful to these forums for helping me learn Pytorch. The reason of doing this in PyTorch was because of recommendation of Professor Chunhua Shen and because of high regards for Pytorch in the scientific community. I can see now after doing this project in Pytorch (in the capacity I could) why it is so highly regarded, and will continue to work in Pytorch in my future projects. Thank you for making me learn Pytorch.

Introduction

Plant Seedling Classification is one of the most popular dataset used for measuring performance of classification algorithms. The idea behind the dataset was that if you can differentiate between a weed and a crop seedling and eliminate the weed, it would lead to better crop yields. It was made by Aarhus University Signal Processing Group in collaboration with University of Southern Denmark. It has over 960 unique plant images belonging to 12 different species at several growth stages.

DataSet

The data is provided to us by two folders named train and test. The train folder contains 12 different sub-folders with each sub-folder belonging to a unique class. Each sub-folder also contains the images of the respective species. The names of the sub-folders are names of the species of bird. This will be directly extracted and used as labels for our images. The dataset can be visualized in the Figure 1. One thing which can be observed is that the dataset is not balanced. Which basically means there are lot of minority classes. This is not generally good for neural networks as it might assume that in real world too there might minority classes as well which won't be the case. I will come back to this in later sections.

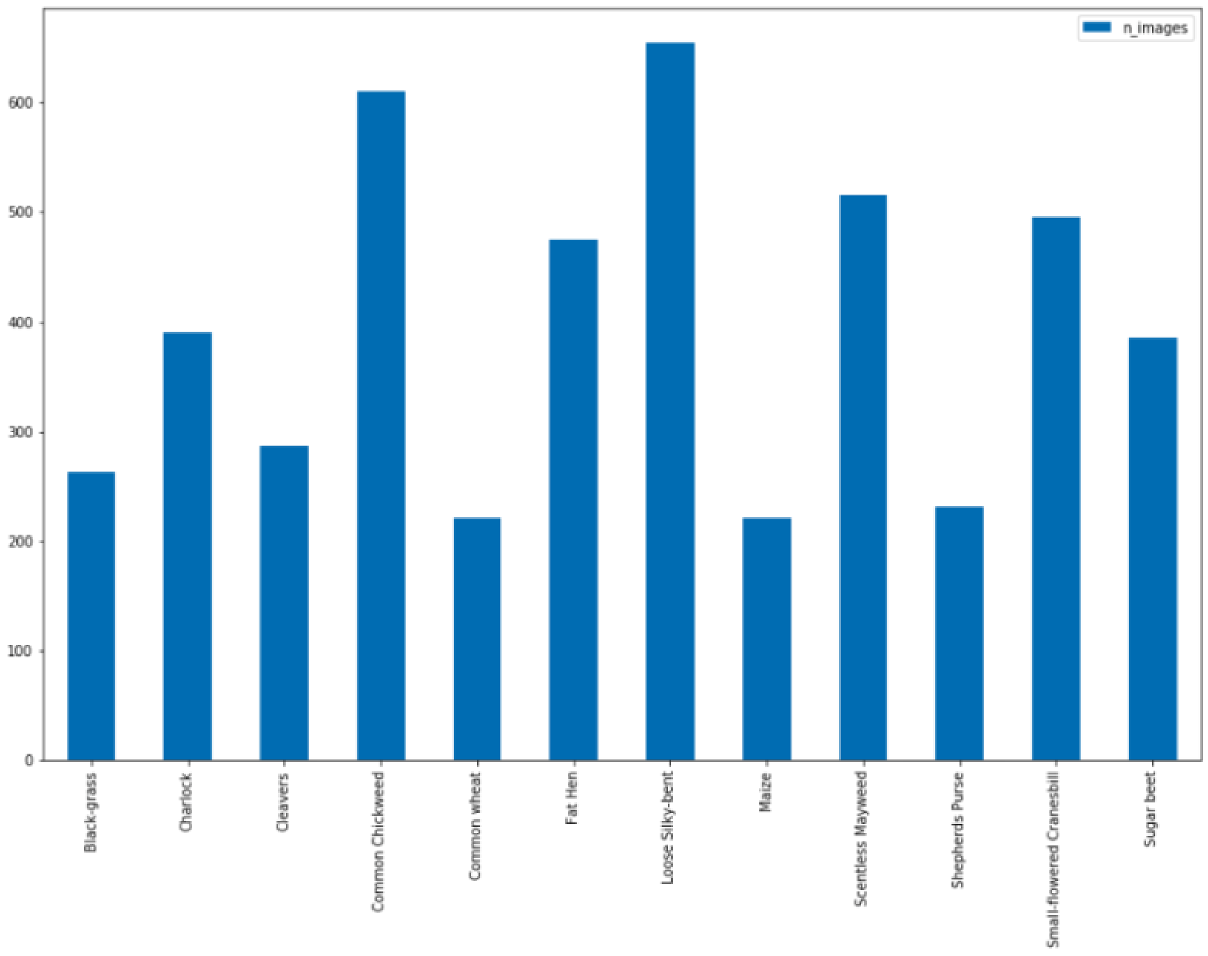
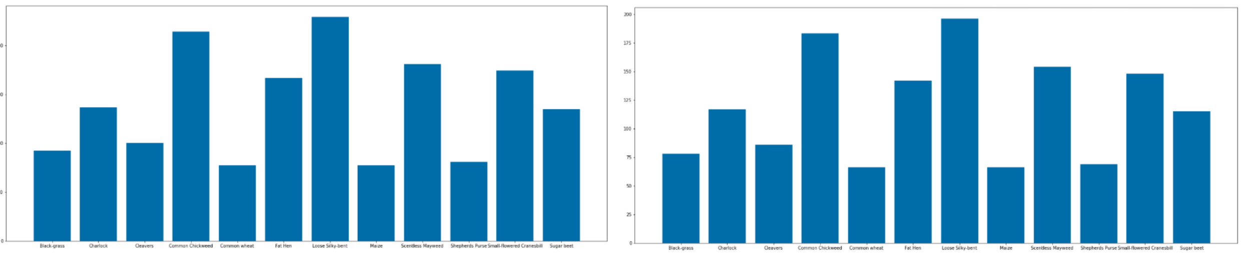


Figure 1: Data Visualization

Data Pre-processing

Now, the data that has been provided to us doesn't have a validation set. We are just given a training folder and a test folder. We need to create a validation set from the training set. Before any modelling, this need to be done to check the performance of our model. It is a good practise in machine learning to separate the given data into training, validation and test set [11]. The training set is the actual dataset that we use to train the model which becomes biases and weights in the case of Neural Network. In this case, i.e. weights and biases will be changed according to the training data [10]. The validation set on the other hand is used to measure the performance of model trained on training data. The results from validation set are used to fine tune our Neural Network. This means that the validation set is not seen by the model on regular basis and is not learning from it. We use the validation set results, and update higher level hyperparameters. So validation set affects our model but it is not doing so directly but indirectly [10]. Now the test set is final stage where we test our finished model. We only come to this step once our model is completely finished. This is the set generally never do any pre-processing on. In this case, the kaggle competition[3] has provided us with test set, and want us to classify and images present in it. The test set is generally well curated. It contains carefully sampled data that spans the various classes that the model would face, when used in the real world [10]. We start by making a new folder in our data directory. It is called new-data. In this sub-directory we will create two new folders called train and validation. Now we will iterate through the original data provided to us and split each class into either train or validation image. For purposes of this research we will split the given data at a ratio, 70 : 30. That is to say that 70% of our data will be used for training and remaining 30% will be used as validation. We can see the splitted data in the Figure 2



(a) Training Data (b) Validation Data

Figure 2: Visualization of the Train and Validation Data-set

Deep Learning Environment

There are two open-source deep learning environment available to us, one is called Keras and other is called PyTorch.

- Keras is a high-level API capable of running on top of TensorFlow, CNTK, Theano, or MXNet (or as tf.contrib within TensorFlow). Since its initial release in March 2015, it has gained favor for its ease of use and syntactic simplicity, facilitating fast development. It's supported by Google[4].
- PyTorch, released in October 2016, is a lower-level API focused on direct work with array expressions. It has gained immense interest in the last year, becoming a preferred solution for academic research, and applications of deep learning requiring optimizing custom expressions. It's supported by Facebook[4].

For the purpose of this project we will be using Pytorch version 1.6.0. PyTorch offers a comparatively lower-level environment for experimentation, giving the user more freedom to write custom layers and look under the hood of numerical optimization tasks

Transformation on the Data-set

Now, because we are working with PyTorch for this project, we need to apply some transformations on the images given to us. As PyTorch doesn't know how to work with images directly. PyTorch need tensors.

A PyTorch Tensor is basically the same as a numpy array: it does not know anything about deep learning or computational graphs or gradients, and is just a generic n-dimensional array to be used for arbitrary numeric computation [8]. $Torchvision.transforms$ contains many such predefined functions and well use the $ToTensor$ transform to convert images into pytorch tensors.

Visualizing Images in the Data-Set

Here we will will see some of the images given in the dataset.

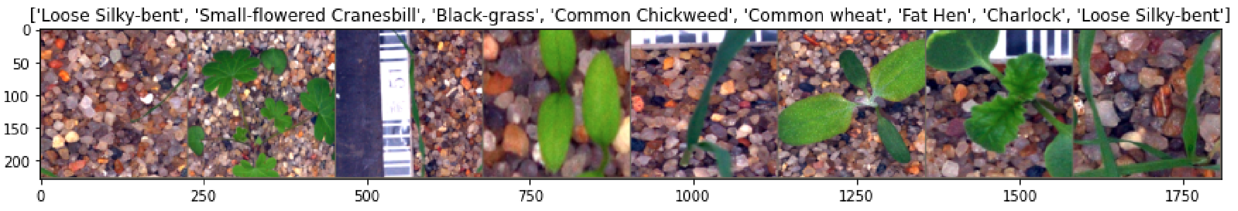

['Loose Silky-bent', 'Small-flowered Cranesbill', 'Black-grass', 'Common Chickweed', 'Common wheat', 'Fat Hen', 'Charlock', 'Loose Silky-bent']
0
50
100
150
200
0
250
500
750
1000
1250
1500
1750

Figure 3: One batch of Images from Training set

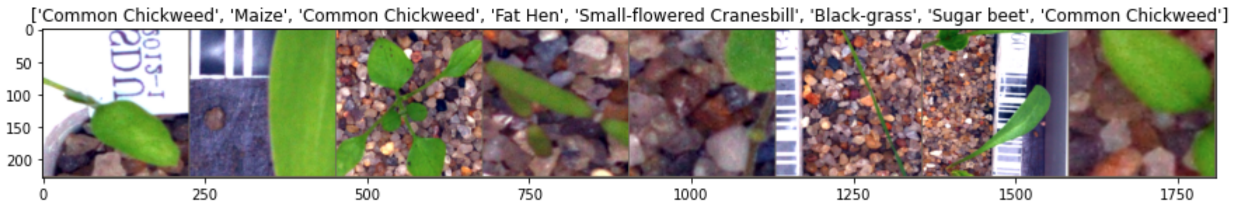

['Common Chickweed', 'Maize', 'Common Chickweed', 'Fat Hen', 'Small-flowered Cranesbill', 'Black-grass', 'Sugar beet', 'Common Chickweed']
0
50
100
150
200
0
250
500
750
1000
1250
1500
1750

Figure 4: Another batch of Images from Training Set

The name of the seedling is given on top of the image. We can see that not all images perfect. Some are blurry, In some cases the seedling in not in the centre. That is why we have to preform necessary transforms on the images. So that that our network can better understand different types of images.

Understanding Neural Networks

What is a Neuron ?

Artificial Neural Networks (ANN) were influenced by what is actually happening in your brain. And although these analogies are very loose, the ANNs have many parallels to their biological 'parent'.

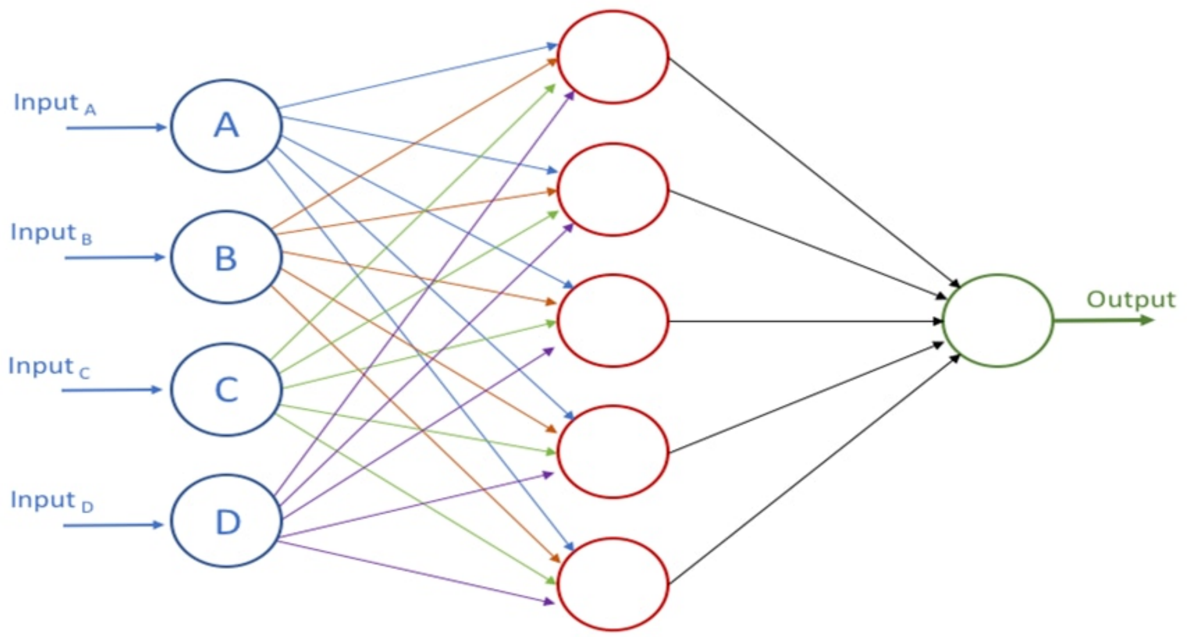


Figure 5: A simple Neural Network

A Neuron is a mathematical function. It takes several numbers as inputs(as many as you want). The neuron in the example given (Figure 5) takes four parameters as input. In our dataset there are 12 different types of seedling, which can be seen as 12 input variable for our machine learning model. For each input neuron, the model is assigned another variable called *weight* of the input. These weights are what makes each neuron unique. They are fixed during testing, but during training these are the numbers we're going to change in order to 'tune' our network

Loss Function

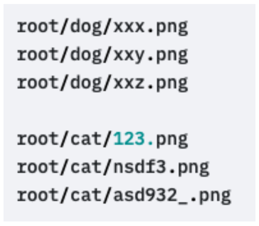
The only thing left to define before I start talking about training is a Loss function. Loss function is a function that tells us, how good our neural network for a certain task. If the Loss function is big then our network doesn't perform very well, we want as small number as possible.

How we train Neural Nets

When we start off with our neural network we initialize our weights randomly. Obviously, it won't give you very good results. In the process of training, we want to start with a bad performing neural network and wind up with network with high accuracy. In terms of loss function, we want our loss function to much lower in the end of training. Improving the network is possible, because we can change its function by adjusting weights. We want to find another function that performs better than the initial one.

PyTorch ImageFolder and DataLoaders

A pytorch ImageFolder function is a generic data loader where the images are arranged in the way shown in Figure 6



```
root/dog/xxx.png
root/dog/xxy.png
root/dog/xxz.png

root/cat/123.png
root/cat/nsdf3.png
root/cat/asd932_.png
```

Figure 6: Root Directory given to ImageFolder Function

A Pytorch DataLoader represents a Python iterable over a dataset, with support for: [6]

- map-style and iterable-style datasets,
- customizing data loading order,
- automatic batching,
- single- and multi-process data loading,
- automatic memory pinning.

In the ImageFolder function we can pass the transforms functions which we want to apply to the datasets such as converting to tensors. After applying this, we call the pytorch dataloader function. The DataLoader has some parameters like $Shuffle, BatchSize$ which basically means that if we want to randomly shuffle the images to the dataloader so that our doesn't learn any pattern based on order of input. Sometimes when you run the model on your machine, you might not have enough memory to load the entire dataset in one go. That's where the concept of batch size came from. One entire datasets is divided into batches. In this case we have taken batch size as 64. One can take any value of the batch size but it is something that does affect model performance to some degree. So we have optimize accordingly.

Making a Neural Network For our Data

PyTorch provides some inbuilt models which we can just initialize and run on our data [7]. For this project I will be making one basic convolutional network and then using models provided by pytorch such as ResNet and VGG.

- **nn.Module** : Base class for all neural network modules.Your models should also subclass this class. Modules can also contain other Modules, allowing to nest them in a tree structure. You can assign the submodules as regular attributes.
- **torch.nn.Sequential**: A sequential container. Modules will be added to it in the order they are passed in the constructor. Alternatively, an ordered dict of modules can also be passed in.
- **Batch Normalization** Batch normalization (also known as batch norm) is a method used to make artificial neural networks faster and more stable through normalization of the input layer by re-centering and re-scaling.
- **Activation Function - ReLU** It stands for rectified linear unit, and is a type of activation function. Mathematically, it is defined as $y = max(0, x)$. ReLU is the most commonly used activation function in neural networks, especially in CNNs. ReLU is linear (identity) for all positive values, and zero for all negative values.
- **Dropout**: During training, randomly zeroes some of the elements of the input tensor with probability p using samples from a Bernoulli distribution. Each channel will be zeroed out independently on every forward call. This has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper "Improving neural networks by preventing co-adaptation of feature detectors"[2].

- **Max Pooling [5]**:

  - Pooling layers provide an approach to down sampling feature maps by summarizing the presence of features in patches of the feature map.
  - Two common pooling methods are average pooling and max pooling that summarize the average presence of a feature and the most activated presence of a feature respectively.
  - Maximum pooling, or max pooling, is a pooling operation that calculates the maximum, or largest, value in each patch of each feature map.
  - The results are down sampled or pooled feature maps that highlight the most present feature in the patch, not the average presence of the feature in the case of average pooling.
  - This has been found to work better in practice than average pooling for computer vision tasks like image classification.

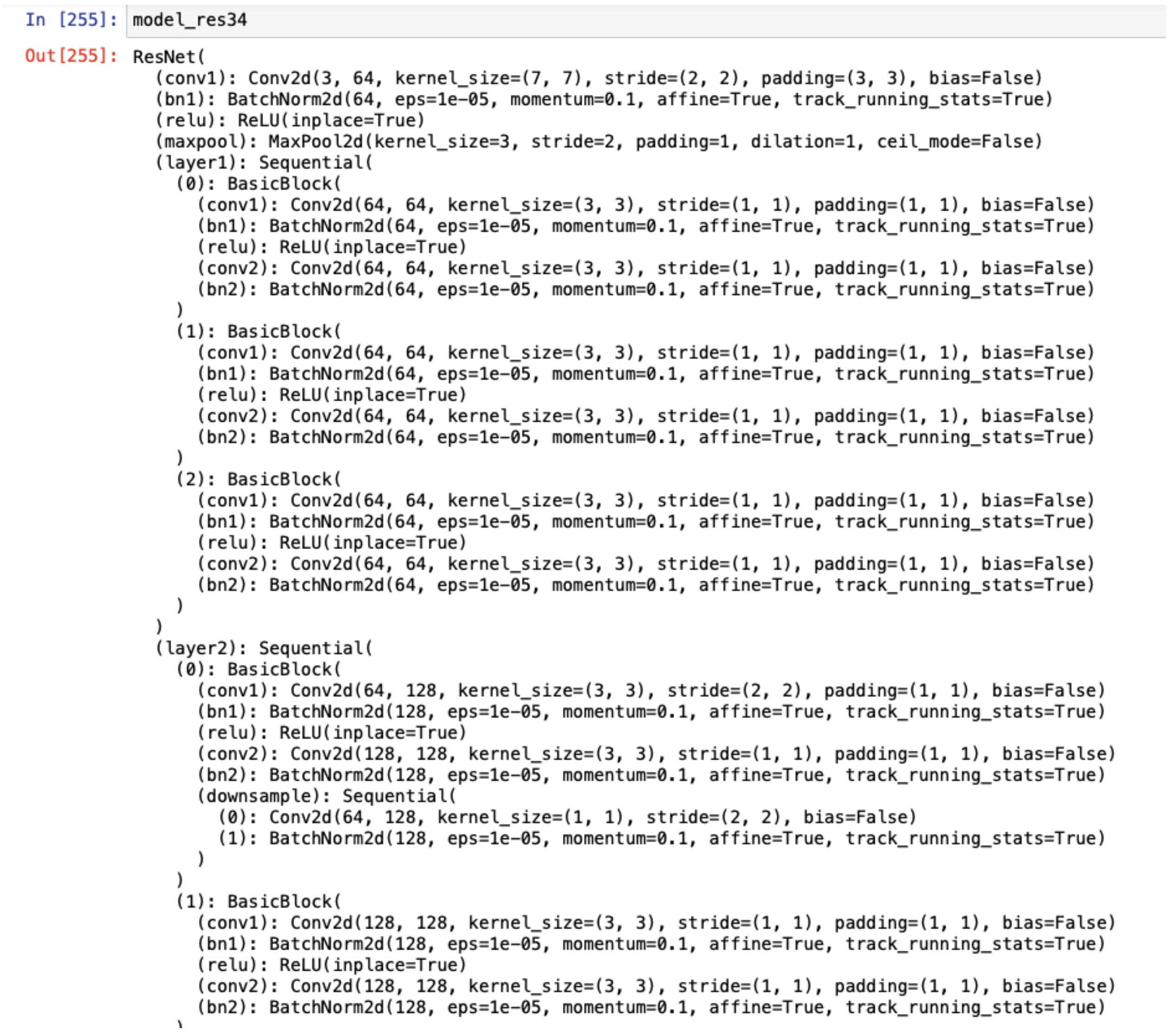
All the neural networks in pytorch are made by using custom modules which are classes inheriting from the nn.Module base class. The advantage of this is that modules can be nested within each other like a tree structure and so you can group different together and keep them logically separate. We created a Net() class which will be inheriting layers from two other class ConvoRes and ConvoCNN with init method defined in both the classes. It gets called when we instantiate the class in Net() which itself has a init function. The forward method is where we combine everything together and it return us the final calculate done by out network.

Training the Network

We start with iterating over the dataloader. Now the dataloader gives us two outputs, one is the images and other being the labels itself. Once we have have image and labels in form of the tensors we pass the image to our defined model. Here all the Convolutional layer, batch normalization and Max pooling takes plance and at its end it's gives us a probability tensor. It is basically a list of 12 probabilities for each of the class. Now we send the output from the network and actual labels to calculate the loss. We are using CrossEntropyLoss for the purpose of this project. Once we calculate the loss, we do the optimization step where it model check how far it is from actual predictions and adjusts its weights. This process keeps on happening according to the epochs we have mentioned.

Pre-trained Models Used

The models which we will use for the classfication purpose are ResNet which is basically "Residual Learning for Image Classfication" [12]. It is a learning framework to ease the training of networks that are substantially deeper than those used previously. This model explicitly reformulate the layers as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions. Two different versions of ResNet are used, One is ResNet34 (Can be seen in Figure 7) and one is ResNet50. The second model which we will use is the called VGG or "Very Deep Convolutional Networks for Large-Scale Image Recognition"[13]. In this model thorough evaluation of networks of increasing depth using an architecture with very small (3x3) convolution filters, which shows that a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16-19 weight layers.



```
In [255]: model_res34

Out[255]: 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)
              )
              (2): 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)
                )
              )
              (1): BasicBlock(
                (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), 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)
```

Figure 7: Structure of a ResNet Network

We can see, the ResNet34 network is very large. It has been meticulously made by researchers based on what works and doesn't work. Be it the stride used or padding used. Stride denotes how many steps we are moving in each steps in convolution.By default it is one. But here we can see the researchers have taken it as two as it works well for classification problems. After taking strides one can observe that the size of output is smaller that input. To maintain the dimension of output as in input , we use padding. Padding is a process of adding zeros to the input matrix symmetrically. We can see in Figure 8 that because of padding the input and output image are of same size [9].

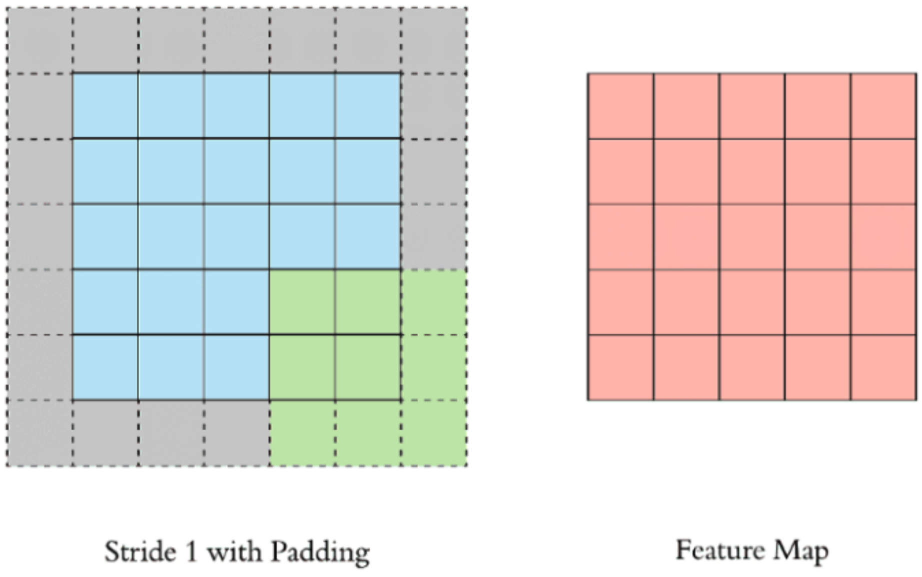
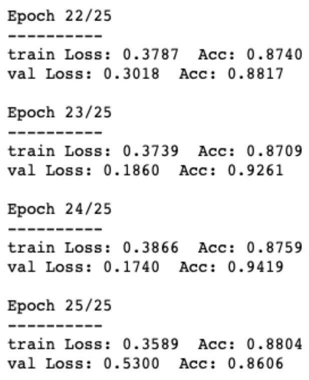


Figure 8: Stride and Padding

Evaluation and results

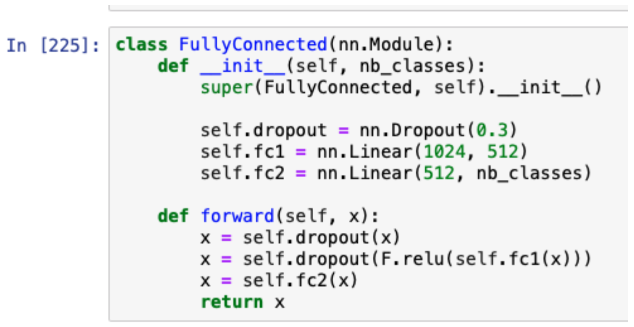
- After training the model for some epochs, which took hours to run on the local machine, we got the best accuracy of our models as 94% on the validation set. You can see the results in figure 9



```
Epoch 22/25
----------
train Loss: 0.3787  Acc: 0.8740
val Loss: 0.3018  Acc: 0.8817

Epoch 23/25
----------
train Loss: 0.3739  Acc: 0.8709
val Loss: 0.1860  Acc: 0.9261

Epoch 24/25
----------
train Loss: 0.3866  Acc: 0.8759
val Loss: 0.1740  Acc: 0.9419

Epoch 25/25
----------
train Loss: 0.3589  Acc: 0.8804
val Loss: 0.5300  Acc: 0.8606
```

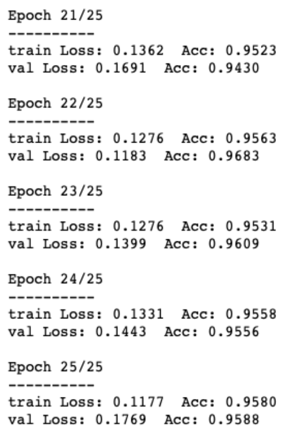
Figure 9: Result From ResNet34 Network

- After doing some review on online resource (Kaggle , Github), I found that combining a fully connected network with ResNet50 gives significantly better results. So I made a small fully connected layer. And combined it with ResNet50 after removing the original ResNet50 fully connected Layer. The new fully connected layer can be seen in figure 10. The results from this were very good.It reached a peak accuracy of around 96% as seen in figure 11



```
In [225]: class FullyConnected(nn.Module):
              def __init__(self, nb_classes):
                  super(FullyConnected, self).__init__()

                  self.dropout = nn.Dropout(0.3)
                  self.fc1 = nn.Linear(1024, 512)
                  self.fc2 = nn.Linear(512, nb_classes)

              def forward(self, x):
                  x = self.dropout(x)
                  x = self.dropout(F.relu(self.fc1(x)))
                  x = self.fc2(x)
                  return x
```

Figure 10: Fully Connected Layer added to ResNet50



```
Epoch 21/25
----------
train Loss: 0.1362  Acc: 0.9523
val Loss: 0.1691  Acc: 0.9430

Epoch 22/25
----------
train Loss: 0.1276  Acc: 0.9563
val Loss: 0.1183  Acc: 0.9683

Epoch 23/25
----------
train Loss: 0.1276  Acc: 0.9531
val Loss: 0.1399  Acc: 0.9609

Epoch 24/25
----------
train Loss: 0.1331  Acc: 0.9558
val Loss: 0.1443  Acc: 0.9556

Epoch 25/25
----------
train Loss: 0.1177  Acc: 0.9580
val Loss: 0.1769  Acc: 0.9588
```

Figure 11: Results from ResNet50

- The results from VGG were not good, that is why they are not shown here. The reason for bad performance of VGG is something i'm still looking for at the time of writing this report.
- I would again like to acknowledge the support of online which was very helpful for the completion of this project.

Future Scope

It can be seen from Figure 1, that training data is infact imbalanced. Imbalance in the training data can mislead the network to learn wrong correlation. We don't want that to happen. That is why it is important to have a balanced dataset. The technique which can be used to balance an imabalanced dataset is called $SMOTE$ which basically stands for "Synthetic Minority Oversampling Technique". One way to achieve this is by simply duplicating examples from the minority class in the training dataset prior to fitting a model. This can balance the class distribution but does not provide any additional information to the model. An improvement on duplicating examples from the minority class is to synthesize new examples from the minority class. This is a type of data augmentation for tabular data and can be very effective. SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line [1].

This is something that can be done in the future on this dataset to achieve even higher accuracy. Because of time constraints I was not able to pursue that, but will continue to work on it in the future.

References

[1] Balancing an imbalanced dataset - https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/.

[2] Dropout- https://pytorch.org/docs/stable/generated/torch.nn.dropout.html.

[3] Kaggle- plant seedling classification - https://www.kaggle.com/c/plant-seedlings-classification/data.

[4] Keras and pytorch - https://deepsense.ai/keras-or-pytorch/.

[5] Max-pooling - https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/.

[6] Pytorch dataloader - https://pytorch.org/docs/stable/data.html.

[7] Pytorch models - https://pytorch.org/docs/stable/torchvision/models.html.

[8] Pytorch tensor - https://pytorch.org/tutorials/beginner/examples$_tensor/two_layer_net_tensor.html$.

[9] Stride and padding - https://towardsdatascience.com/covolutional-neural-network-cb0883dd6529.

[10] Train, validation and test sets - https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7.

[11] Aurélien Géron. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems.* O'Reilly Media, 2019.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[13] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.