Cassava Leaf Disease Classification

Machine Learning Project

Presented to: Prof. Tanmoy Sir

By: Riya Parikh

222016007

Introduction

- Cassava Shrub is a root vegetable that grows underground. It has the Latin name Manihot esculenta.
- Cassava are similar to sweet potatoes and its leaves are also edible.
- It is a food security crop for the majority of people in Sub-Saharan Africa and is the second largest source of Carbohydrates, and thus cultivated by almost 80% of farms.

Problem Statement

- The crop yields are hampered by viral diseases prevalent in them and is the major cause of it.
- The farmers are still dependent on the old methods of a visual inspection of the plant by agriculture experts for deciding on the disease's prevalence and are highly dependent on government funding for agriculture.
- All this inspection requires high skill labor which is scarce in the region and is very costly and time consuming processes.

Objective

- To reduce ailment of Sub-Saharan African farmers by machine learning methods for classification of diseases occurring in the cassava plants leaf.
- The task of the project will be to classify cassava leaf images into four disease categories for farmers to recognize afflict plants early on.

About Datasets

- The datasets collected are labelled images of cassava leaf collected during a survey in Uganda by National Resource Research Institute (NaCRRI) in collaboration with Makerere University, Kampala lab for AI.
- It is a compilation of the most realistic data set that farmers could provide in real life for diagnosis.
- There are images collected from farmers and diagnosed by experts at National Crops Resources Research Institute (NaCRRI) in collaboration with the AI lab at Makerere University, Kampala.
- They are Open to use on Kaggle.

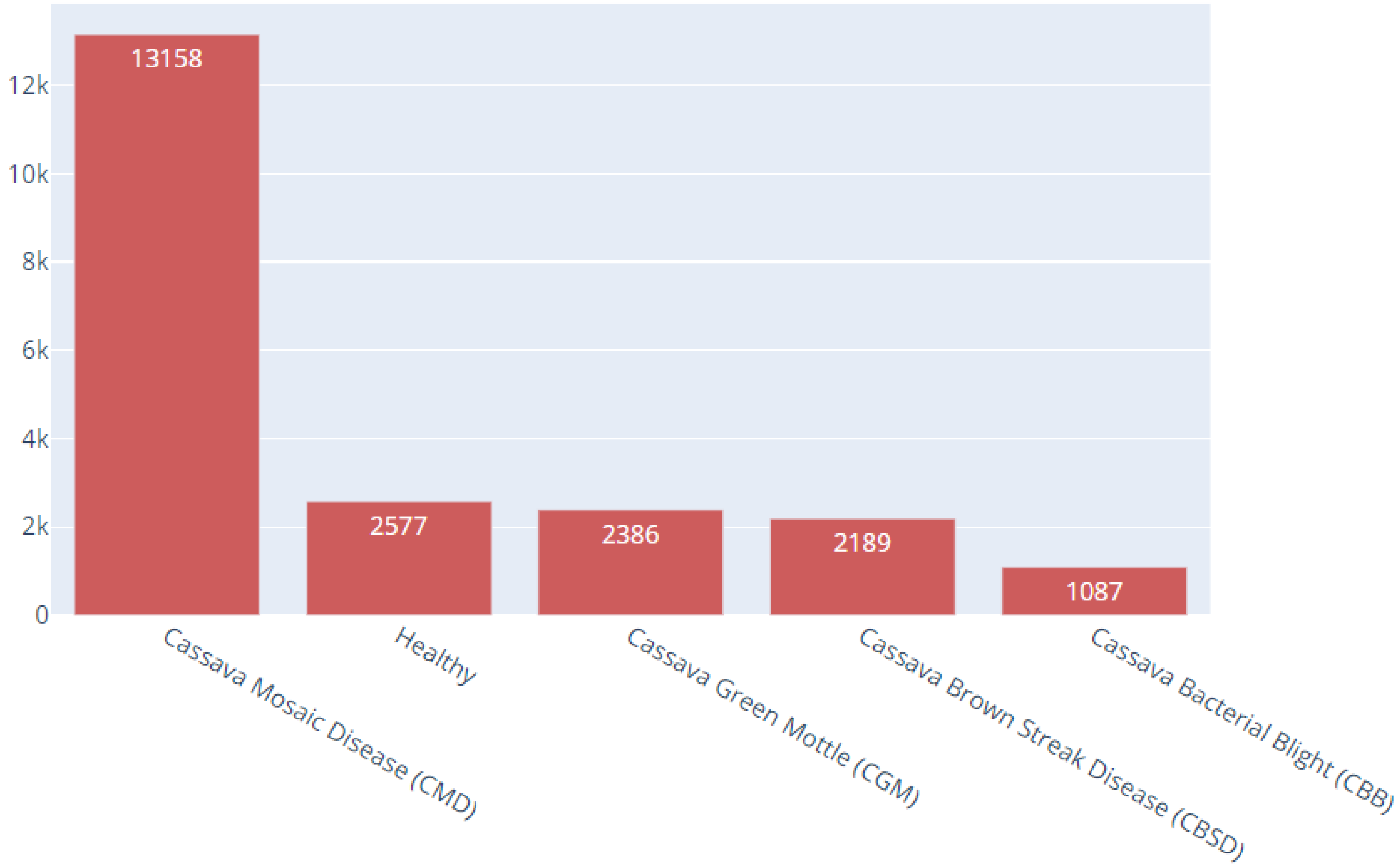
Understanding the Data

```json
{
    "0": "Cassava Bacterial Blight (CBB)",
    "1": "Cassava Brown Streak Disease (CBSD)",
    "2": "Cassava Green Mottle (CGM)",
    "3": "Cassava Mosaic Disease (CMD)",
    "4": "Healthy"
}
```

Understanding the Data

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

Understanding the Data



Understanding the Data

Approach

- As the data in hand is image data
- Robust machine learning methods best suited for image to employed
- Features need to extracted from large data
- Neural Networks Best suited for task
- Among Neural Networks, Convolutional Neural Networks best suited for Computer Vision application.

Train Test Split

```
train = int(len(df) * 0.7)
valid = int(len(df) * 0.2)
test = len(df)-train-valid
```

Converting Images into required format

```
#Preprossesing the data into image and label
class CassavaDataset(Dataset):
    def __init__(self, dataframe, transform = None):
        super().__init__()
        self.df = dataframe
        self.transform = transform

    def __len__(self):
        return len(self.df["path"])

    def __getitem__(self, index):
        # get path and label
        path = self.df["path"][index]
        label = self.df["label"][index]
        # load image
        with open(path, 'rb') as f:
            image = Image.open(f)
            image = image.convert("RGB")
        # transform the image
        if self.transform is not None:
            image = self.transform(image)

        return image, label
```

After Transformation

```
train_data[0]
```

```
(tensor([[[-1.5357, -1.5357, -1.5528,  ..., -1.3130, -1.2617, -1.2103],
          [-1.5185, -1.5185, -1.5185,  ..., -1.2445, -1.2445, -1.2103],
          [-1.5185, -1.5014, -1.5014,  ..., -1.2103, -1.1760, -1.1589],
          ...,
          [-1.3473, -1.3815, -1.4158,  ..., -1.0219, -1.2617, -1.3130],
          [-1.3644, -1.4500, -1.4158,  ..., -1.1075, -1.3302, -1.3815],
          [-1.3302, -1.3644, -1.4500,  ..., -1.1075, -1.3473, -1.4158]],

         [[ 0.9580,  0.9580,  0.9755,  ..., -0.2325, -0.1275, -0.0574],
          [ 0.9755,  0.9930,  1.0105,  ..., -0.0924, -0.0574, -0.0049],
          [ 0.9755,  0.9930,  1.0280,  ..., -0.0049,  0.0301,  0.0476],
          ...,
          [ 0.6954,  0.6604,  0.6954,  ...,  1.2556,  1.0980,  1.0105],
          [ 0.6954,  0.6429,  0.6954,  ...,  1.1856,  1.0280,  0.9580],
          [ 0.7654,  0.7479,  0.6954,  ...,  1.1856,  1.0105,  0.9230]],

         [[ 0.7228,  0.7228,  0.7402,  ..., -1.2119, -1.3513, -1.4210],
          [ 0.7402,  0.7576,  0.7751,  ..., -1.2119, -1.3513, -1.4036],
          [ 0.7402,  0.7576,  0.7751,  ..., -1.2990, -1.3164, -1.3339],
          ...,
          [ 0.3742,  0.3568,  0.3568,  ...,  1.1062,  0.9842,  0.9668],
          [ 0.3916,  0.3045,  0.3219,  ...,  0.9668,  0.8448,  0.8274],
          [ 0.4439,  0.3568,  0.2348,  ...,  0.8971,  0.7576,  0.7228]]]), 3)
```

Why CNN?

- Good in processing data that has a grid pattern, such as images.
- designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns.

Design

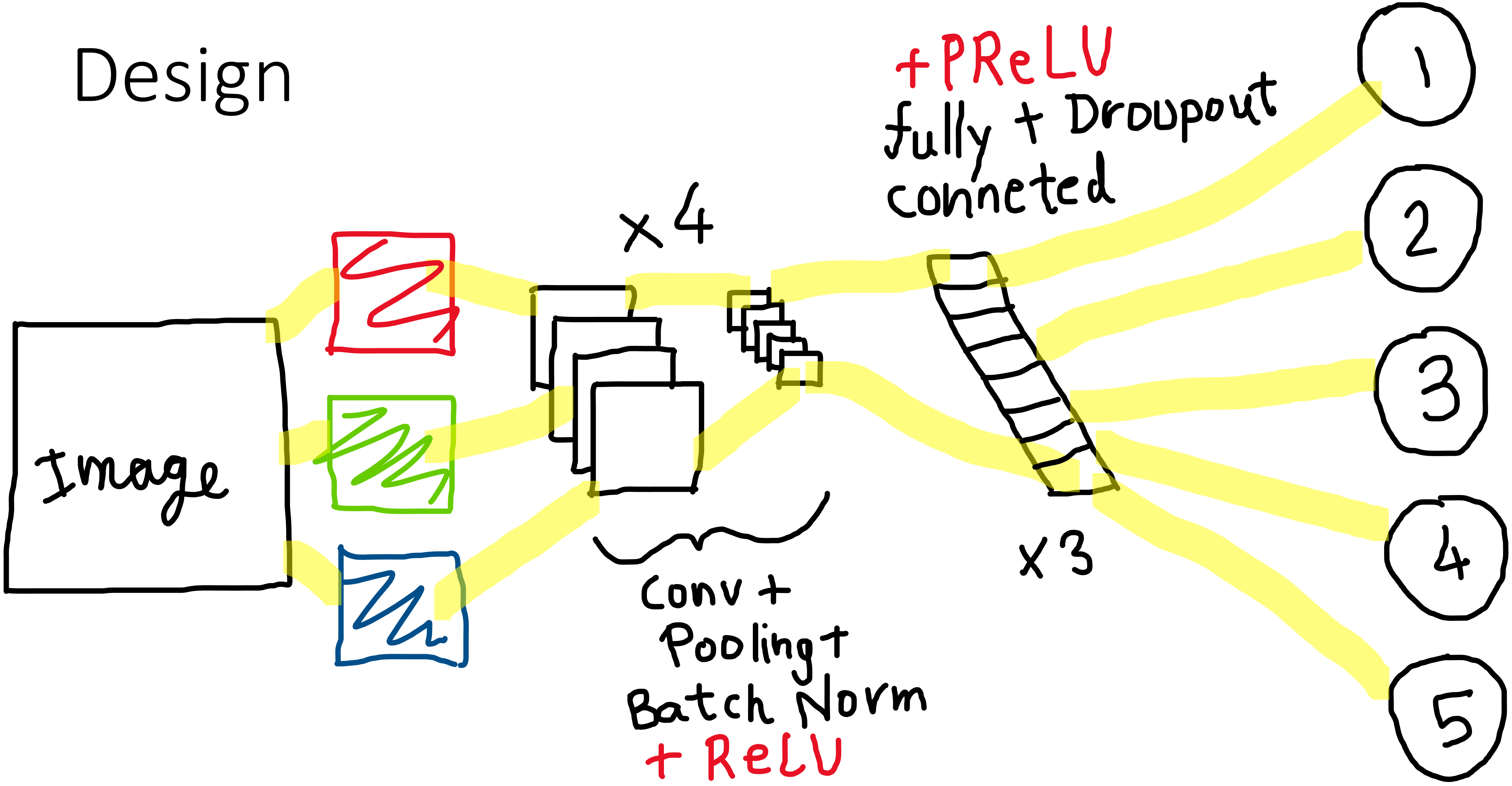


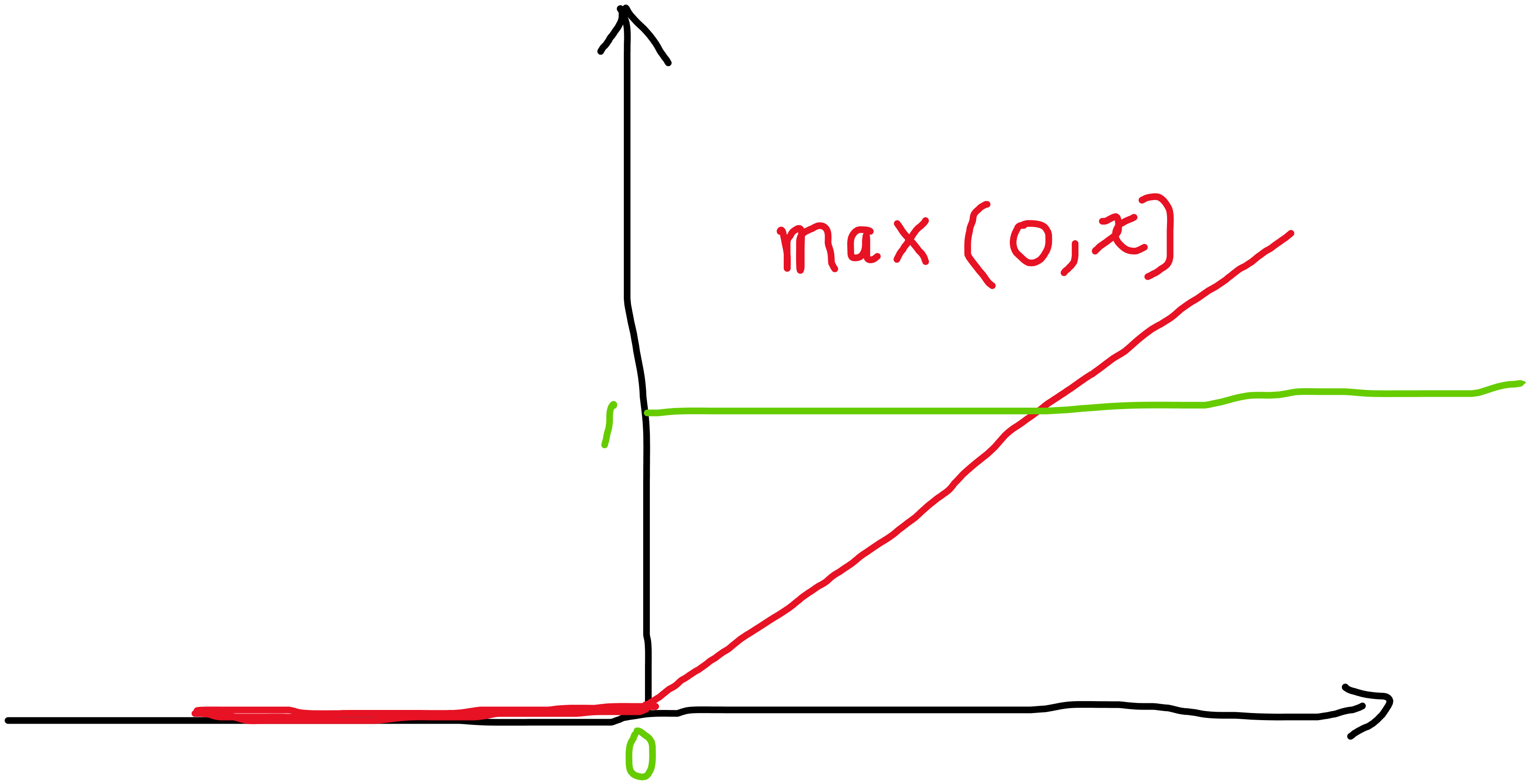
Image
x4
Conv +
Pooling +
Batch Norm
+ ReLU
+ PReLU
fully + Droupout
conneted
x 3
1
2
3
4
5

Structure of CNN used

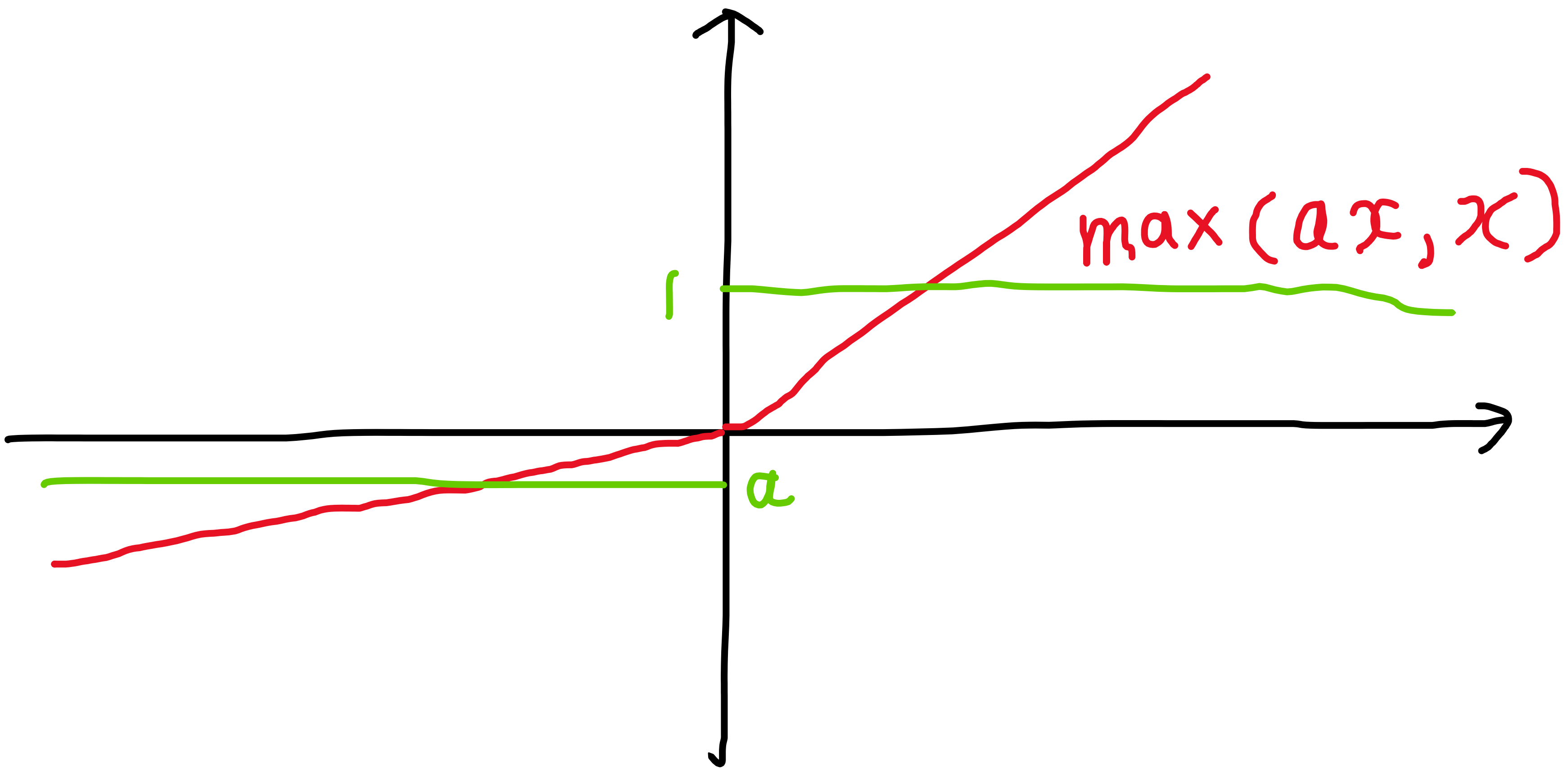
```
#convolutional neural network
class Leaf_CNN(nn.Module):
    def __init__(self):
      super(Leaf_CNN, self).__init__()

      self.cnn_layers = nn.Sequential(

        #defining a 2D convolution layer 1
        nn.Conv2d(in_channels = 3, out_channels = 64, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(64),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2,2),

        #defining a 2D convolution layer 2
        nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(128),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2,2),

        #defining a 2D convolution layer 3
        nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(256),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2,2),

        #defining a 2D convolution layer 4
        nn.Conv2d(in_channels = 256, out_channels = 512, kernel_size=3, stride=1, padding=1),
        nn.BatchNorm2d(512),
        nn.ReLU(inplace=True),
        nn.MaxPool2d(2,2),

      )
```

```
      self.linear_layers = nn.Sequential(
            nn.Linear(512, 16 * 16),
            nn.PReLU(),
            nn.Dropout(0.2, inplace=True),
            nn.Linear(256, 8 * 8),
            nn.PReLU(),
            nn.Dropout(0.2, inplace=True),
            nn.Linear(64, 5),
            )

    def forward(self, x):
        x = self.cnn_layers(x)
        x = torch.mean(x, dim = 3)
        x, _ = torch.max(x, dim = 2)
        x = self.linear_layers(x)
        return x
```

Why Activation function – ReLU is used



Why Activation function – PReLU is used



Batch Normalization

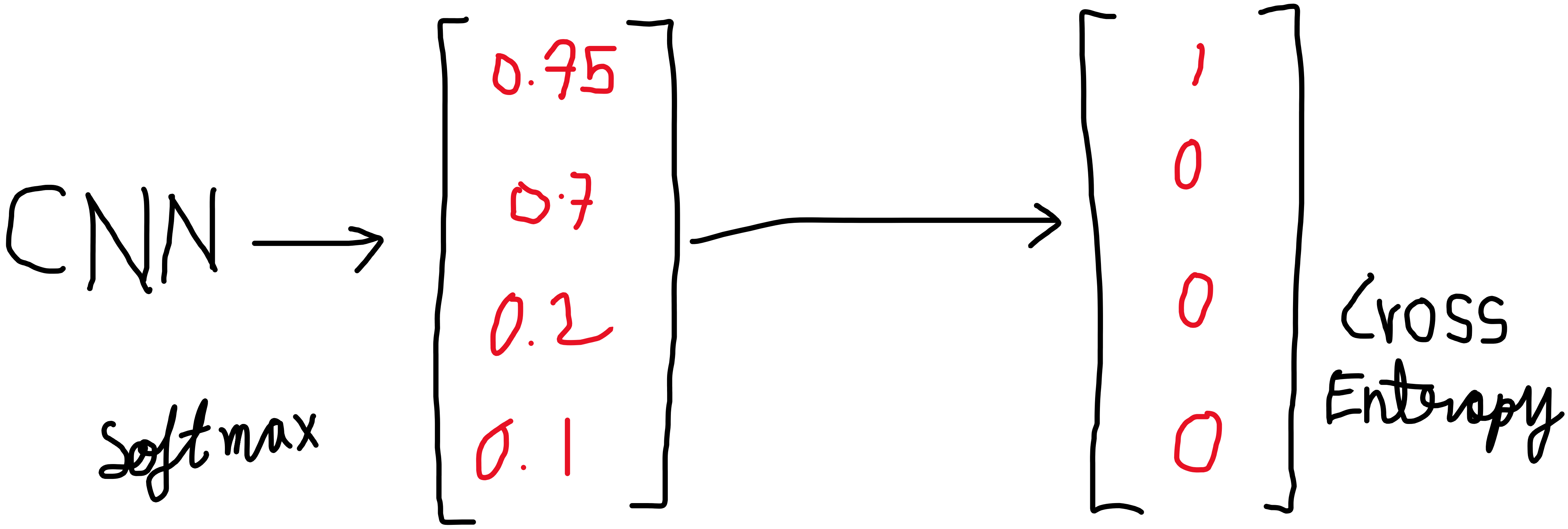
- Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch.
- This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

Why Optimizer – Adam is used

- first-order gradient-based optimization of stochastic objective functions
- The method is straightforward to implement
- Is computationally efficient
- Has little memory requirements
- Is invariant to diagonal rescaling of the gradients
- Is well suited for problems that are large in terms of data and/or parameters.

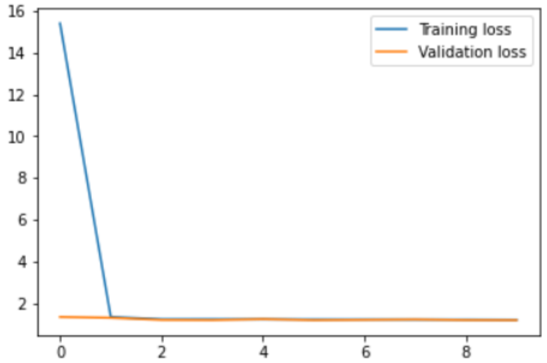
```
optim = torch.optim.Adam(model.parameters(), lr = lr)
```

Cross Entropy Loss



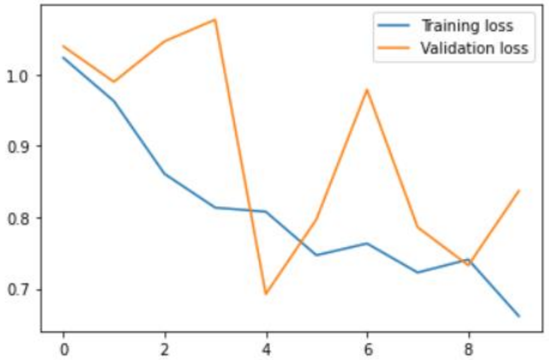

```
criterion = nn.CrossEntropyLoss()
```

Progress over the model



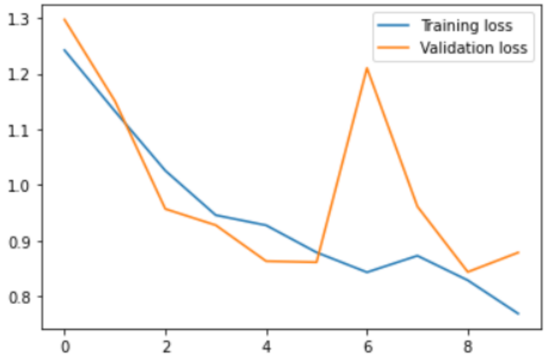
0.6383177570093458

```
num_epoch = 10
num_classes = 5
batch_size = 64
lr = 0.1
```



0.6789719626168225

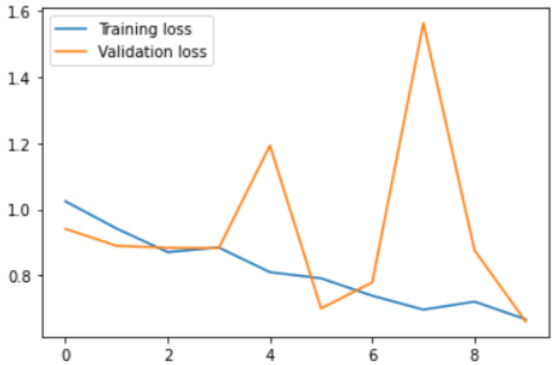
```
num_epoch = 10
num_classes = 5
batch_size = 64
lr = 0.001
```



0.6682242990654206

```
num_epoch = 10
num_classes = 5
batch_size = 64
lr = 0.01
```

Dropout 0.3



0.746261682242990б

```
num_epoch = 10
num_classes = 5
batch_size = 64
lr = 0.001
```

The optimal parameters and Accuracy

```
num_epoch = 10
num_classes = 5
batch_size = 64
lr = 0.001
```

Dropout 0.2

Time: 8736.145	Epoch: 1	Training Loss: 1.004	Validation Loss: 1.200744	Acc: 0.57
Time: 512.358	Epoch: 2	Training Loss: 0.882	Validation Loss: 1.361476	Acc: 0.61
Time: 503.401	Epoch: 3	Training Loss: 0.815	Validation Loss: 0.881436	Acc: 0.63
Time: 505.876	Epoch: 4	Training Loss: 0.781	Validation Loss: 0.855999	Acc: 0.65
Time: 497.903	Epoch: 5	Training Loss: 0.726	Validation Loss: 0.946320	Acc: 0.66
Time: 494.087	Epoch: 6	Training Loss: 0.706	Validation Loss: 0.699270	Acc: 0.67
Time: 498.044	Epoch: 7	Training Loss: 0.657	Validation Loss: 0.789187	Acc: 0.68
Time: 488.407	Epoch: 8	Training Loss: 0.724	Validation Loss: 0.710640	Acc: 0.69
Time: 489.352	Epoch: 9	Training Loss: 0.662	Validation Loss: 0.803916	Acc: 0.69
Time: 485.247	Epoch: 10	Training Loss: 0.602	Validation Loss: 0.610595	Acc: 0.70

0.7542056074766356

Forward Path

- Check possibility of further increase in accuracy.
- To increase accuracy up to 80%.
    - Analyse the current model
    - Check for increase in number of layers
    - Check for increasing in dimensions of the layers
    - Check for methods like auto weight updating
    - Exploring the new Models and their approach.

Thank You