CASSAVA LEAF DISEASE CLASSIFICATION

Tushar Virk (z5208092)

UNSW Sydney

COMP9417 MACHINE LEARNING PROJECT

1. Introduction

Cassava is a starchy root vegetable, with the tuber part of it underground and the shrub above ground [1]. Due to its ability to endure harsh conditions, the cassava plant is a crucial crop that is one of the largest providers of carbohydrates in Africa. There is a large importance on ensuring that these crops are largely disease free, making them readily available for consumption. A major part of this is the detection of viral diseases from a visual analysis of cassava leaves.

The Cassava Leaf Disease Classification challenge on Kaggle calls for the utilisation of machine learning, computer vision, etc. to tackle the problem of disease detection. Specifically, within this challenge, competitors are asked to implement a model that takes an image of a cassava leaf and predicts it's disease classification label. The dataset itself was largely crowdsourced from farmers taking photos, with labels being provided by experts. The Kaggle description of the project suggests that the format of this dataset is one that "most realistically represents what farmers would need to diagnose in real life" [2].

The best way forward from this would be to utilise deep learning, specifically convolutional neural networks (CNNs) as their inception was inspired by findings in studies of biological vision [3], making it a very suitable technique for image classification.

2. Background

2.1 Convolutional Neural Networks (CNNs)

CNNs are deep learning algorithms most known for their proficiency in image classification tasks (although they can also be used for other classification tasks). I will be discussing CNNs within the context of image classification tasks. A key benefit of using CNNs are their ability to automatically extract and assign importance (weights, biases, etc.) to features from input images [5], in other words there is very minimal (sometimes none) pre-processing required for input images as the CNN will eventually learn the appropriate filters that extract important classification features. Before diving into CNNs, we must first understand what convolution operations and pooling are.

2.1.1 Convolution Operation

Elements of a convolution operation.

Input image (width, height, number of channels [RGB = 3, grayscale = 1]) – the image to perform the convolution operation on

Kernel (filter) (width, height) – the grid of weights that make up our filter/kernel, representing the feature it is trying to extract from the input image

Stride (number) – the number of steps the kernel takes when convolving over the input image (a stride of 1 means non-stride)

Padding:

- Valid padding – apply the kernel without padding, for example a 5x5x1 image after convolution with a kernel of size 3x3x1 has the shape 3x3x1.
- Same padding – zero pad around the image and then apply the kernel, for example a 5x5x1 image after same padding will have shape 6x6x1 and after convolution with a kernel of size 3x3x1 it will have a shape of 5x5x1

Now, we will look at a simple convolution operation on a 5x5x1 image with a 3x3x1 kernel, we will use a stride of 1 and valid padding.

Note that the kernel doesn't need to be a square.

The image matrix is as follows:

$$I = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

The kernel matrix is as follows:

$$K = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

We start at the top left of matrix I and take the first 3x3x1 section of it, let's call it $I^{(1)}$

$$I^{(1)} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

We index $I^{(1)}$ like $I^{(1)}_{\text{row,column}}$, for example $I^{(1)}_{1,1} = 1$, $I^{(1)}_{1,2} = 1$, and so on. We similarly index $K_{\text{row,column}}$. Now we can define the convolution operation for rows m by columns n:

$$\text{Conv}(K, I) = \sum_{i=1}^{m} \sum_{j=1}^{n} K_{i,j} \cdot I_{i,j}$$

For our example, we have:

$$\text{Conv}\left(K, I^{(1)}\right) = \sum_{i=1}^{3} \sum_{j=1}^{3} K_{i,j} \cdot I^{(1)}_{i,j}$$

$$= [(1 \cdot 1) + (0 \cdot 1) + (1 \cdot 1)] + [(0 \cdot 0) + (1 \cdot 1) + (0 \cdot 1)] + [(1 \cdot 0) + (0 \cdot 0) + (1 \cdot 1)]$$

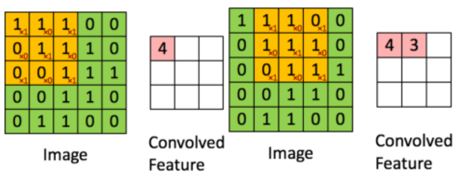
$$= 4$$

Similarly, $I^{(2)} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}$ and $\text{Conv}(K, I^{(2)}) = 3$, and so on. In general, the kth output of the convolution operation is:

$$\text{Out}(k) = \text{Conv}(K, I^{(k)})$$

Below you can find a demonstration of the entire process.

Fig 1.2 Convolution example [5]



The output shape is then $\left(1+\frac{J-M}{s}\right) \times \left(1+\frac{K-N}{s}\right) \times 1$ for an input image of size $J \times K$, a kernel of size $M \times N$ and stride s.

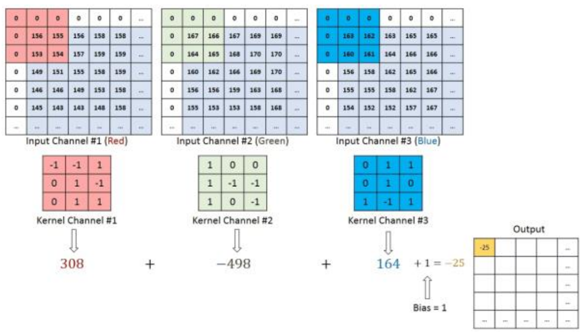
The process for an image with 3 channels (for example RGB) is the same instead there are also 3 channels for the kernel whose weights aren't necessarily the same.

Let I be an input image with 3 channels (RGB). Then I_R represents the red channel, I_G the green channel and I_B the blue channel. Let K be our kernel with 3 channels, then in a similar fashion we define K_R, K_G and K_B. Finally, let b be some bias term. Now our output at the kth convolution operation would be:

$$\text{Out}(k) = \text{Conv}(K_R, I_R^{(k)}) + \text{Conv}(K_G, I_R^{(k)}) + \text{Conv}(K_B, I_B^{(k)}) + b$$

Below you can find a demonstration of the process for a 3-channel image.

Fig 1.3 3-channel convolution example [5]



This motivates the usage of CNNs for image classification tasks instead of simply flattening input images and passing it into a regular neural network as the kernels help to preserve temporal and spatial dependencies within an image [5], whereas this information can be lost or less important when the image input is flattened first.

2.1.2 Pooling

Pooling is like a convolution operation in the sense that it reduces the dimensionality of the input, and each element of the output is due to some operation on values overlapping with a kernel or filter. There are two types of pooling:

Max pooling. Take the max value of all values overlapping with the kernel We define max pooling as

$$\text{MaxPool}(m, n, I) = \max\left\{I_{i,j}\ \forall i \in \{1..m\}\ \forall j \in \{1..n\}\right\}$$

Where $m \times n$ = kernel dimensions. In general, for the kth output we have:

$$\text{Out}(m, n, k) = \text{MaxPool}(m, n, I^{(k)})$$

Average pooling. Take the average value of all values overlapping with the kernel We define average pooling as:

$$\text{AvgPool}(m, n, I) = \frac{1}{m \times n} \sum_{i=1}^{m} \sum_{j=1}^{n} I_{i,j}$$

In general, for the kth output we have:

$$\text{Out}(m, n, k) = \text{AvgPool}(m, n, I^{(k)})$$

Note that stride can also apply when pooling. In the examples below we will assume a stride of 2.

Say we have input image:

$$I = \begin{bmatrix} 10 & 11 & 68 & 98 \\ 37 & 17 & 23 & 32 \\ 90 & 19 & 43 & 55 \\ 45 & 20 & 55 & 77 \end{bmatrix}$$

And the kernel or filter is a 2x2 square. Then:

$$I^{(1)} = \begin{bmatrix} 10 & 11 \\ 37 & 17 \end{bmatrix}, I^{(2)} = \begin{bmatrix} 68 & 98 \\ 23 & 32 \end{bmatrix}, I^{(3)} = \begin{bmatrix} 90 & 19 \\ 45 & 20 \end{bmatrix}, I^{(4)} = \begin{bmatrix} 43 & 55 \\ 55 & 77 \end{bmatrix}$$

Then for max pooling we have:

$$\text{Out}(1) = \text{MaxPool}(2,2,I^{(1)}) = \max\{10,11,37,17\} = 37$$

$$\text{Out}(2) = \text{MaxPool}(2,2,I^{(2)}) = \max\{68,98,23,32\} = 98$$

$$\text{Out}(3) = \text{MaxPool}(2,2,I^{(3)}) = \max\{90,19,45,20\} = 90$$

$$\text{Out}(4) = \text{MaxPool}(2,2,I^{(4)}) = \max\{43,55,55,77\} = 77$$

Our final output for max pooling would be:

$$\begin{bmatrix} 37 & 98 \\ 90 & 77 \end{bmatrix}$$

For average pooling we have:

$$\text{Out}(1) = \text{AvgPool}(2,2,I^{(1)}) = \frac{1}{2 \times 2}(10 + 11 + 37 + 17) = 18.75$$

$$\text{Out}(2) = \text{AvgPool}(2,2,I^{(2)}) = \frac{1}{2 \times 2}(68 + 98 + 23 + 32) = 55.25$$

$$\text{Out}(3) = \text{AvgPool}(2,2,I^{(3)}) = \frac{1}{2 \times 2}(90 + 19 + 45 + 20) = 43.50$$

$$\text{Out}(4) = \text{AvgPool}(2,2,I^{(4)}) = \frac{1}{2 \times 2}(43 + 55 + 55 + 77) = 57.50$$

Our final output for average pooling would be:

$$\begin{bmatrix} 18.75 & 43.50 \\ 55.25 & 57.50 \end{bmatrix}$$

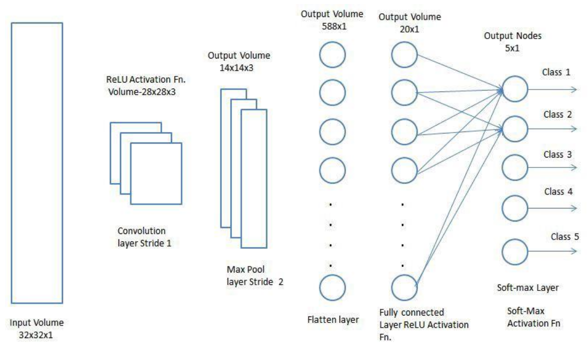
Max pooling helps in the identification of dominant features. In general, max pooling is the preferred option.

2.1.3 Putting it all together

A simple network will follow this structure:

$$\text{Input} \rightarrow \text{Conv} \xrightarrow{\text{ReLU}} \text{Pooling} \rightarrow \text{Fully Connected} \xrightarrow{\text{SoftMax}} \text{Output}$$

Fig 1.4 Diagram of full simple CNN [5]



To add complexity and for the extraction of more complex features, CNNs are sometimes stacked for more complex classification tasks. Such architectures are composed of blocks:

$$\text{Block} = \text{Conv} \xrightarrow{f} \text{Pooling}$$

$$\text{Input} \rightarrow [\text{Block}_1] \rightarrow [\text{Block}_2] \rightarrow \cdots \rightarrow [\text{Block}_n] \rightarrow \text{Fully Connected} \rightarrow \text{Output}$$

In the above f represents an activation function such as ReLU, tanh, etc.

Adding Blocks in the above fashion increases the depth of the network and therefore enriches the features extracted by the model. Many image classification tasks greatly benefit from very deep networks, particularly noted in implementations for the ImageNet [6] dataset. However, a degradation problem is introduced as model depth is increased, where accuracy degrades further as more layers are added to an already deep model [7].

2.1.4 Residual Networks

The ResNet model aims to tackle this problem through the hypothesis that it is easier to optimize a residual mapping than the original mapping [7]. In this case the deeply stacked network is the original mapping denoted:

$$\mathcal{H}(\mathbf{x})$$

Where $\mathbf{x}$ represents the inputs into the network. Now we let

$$\mathcal{F}(\mathbf{x}) = \mathcal{H}(\mathbf{x}) - \mathbf{x}$$

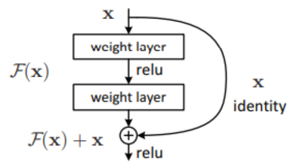
Denote the residual mapping. We can rewrite:

$$\mathcal{H}(\mathbf{x}) = \mathcal{F}(\mathbf{x}) + \mathbf{x}$$

This formulation for our desired mapping $\mathcal{H}(\mathbf{x})$ can be realized in our implementations using shortcut or skip connections (connections in the network which bypass or "skip" one or more layers). In the case of ResNet, these skip connections will simply perform the identity mapping:

$$I(\mathbf{x}) = \mathbf{x}$$

Fig 1.5 Representation of the residual mapping utilizing skip connections [7]



x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 2. Residual learning: a building block.

Through experimentation, it was shown that deep ResNets were easy to optimize whereas the non-residual counterpart networks suffered higher training error as depth increased. Furthermore, ResNets benefitted from accuracy gains as depth increased [7]. ResNet models have the following architecture:

$$\text{Input} \rightarrow \text{Conv} \rightarrow \text{Batch Norm} \xrightarrow{\text{ReLU}} \text{Max Pool} \rightarrow \left[\text{Layer}_1^{(n_1)}\right] \rightarrow \cdots \rightarrow \left[\text{Layer}_4^{(n_4)}\right] \\ \rightarrow \text{AAP} \rightarrow \text{Linear} \rightarrow \text{Output}$$

$$\text{Layer}^{(n)} = [\text{Block}_1|\text{BigBlock}_1] \rightarrow \cdots \rightarrow [\text{Block}_n|\text{BigBlock}_n] \rightarrow$$

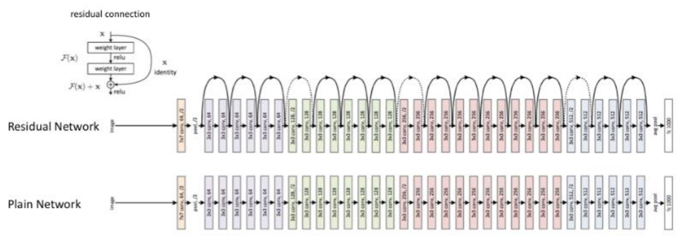
$$\text{Block} = (\mathbf{x})\ \text{Conv} \rightarrow \text{Batch Norm} \xrightarrow{\text{ReLU}} \text{Conv} \rightarrow \text{Batch Norm} \xrightarrow{\text{IDS}} (+\mathbf{x}) \xrightarrow{\text{ReLU}}$$

$$\text{BigBlock} = (\mathbf{x})\ \text{Conv} \rightarrow \text{Batch Norm} \xrightarrow{\text{ReLU}} \text{Conv} \rightarrow \text{Batch Norm} \xrightarrow{\text{ReLU}} \text{Conv} \\ \rightarrow \text{Batch Norm} \xrightarrow{\text{IDS}} (+\mathbf{x}) \xrightarrow{\text{ReLU}}$$

IDS represents identity down-sampling on initial input $\mathbf{x}$ where $\text{IDS} = \text{Conv} \rightarrow \text{Batch Norm} \rightarrow$, applied on the initial input $\mathbf{x}$ only if initial input $\mathbf{x}$ is not of the same shape as the current modified $\mathbf{x}$.

$(+\mathbf{x})$ represents the identity skip connection where the initial $\mathbf{x}$ is simply added to the modified $\mathbf{x}$.

Fig 1.6 ResNet architecture (34 layer) compared to plain stacked 34 layers [7]. The solid line jumps indicate identity skips and the dotted line jumps indicate identity skips but down-sampling needs to be applied since initial $\mathbf{x}$ will be of a different shape.



2.2 Other Techniques/Methods

2.2.1 Batch Normalization

Batch normalization is a technique used to tackle the problem of constantly changing distributions of network inputs (internal covariate shift) in deep layers after weight updates. We standardize/normalize the input of each mini batch $\mathcal{B}$ (i.e. scale data such that mean $\mu_{\mathcal{B}} = 0$ and standard deviation $\sigma_{\mathcal{B}} = 1$). This allows for the mitigation of oscillating means and variances across different batches which impacts model optimization. The pytorch implementation of batch normalization performs the following for each input element of a mini batch $\mathcal{B}$ [8].

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

Where:

- $\mathrm{E}[x] = \mu_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|} x_i$ is the mini batch mean
- $\mathrm{Var}[x] = \sigma_{\mathcal{B}}^2 = \frac{1}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|} (x_i - \mu_{\mathcal{B}})^2$ is the mini batch variance
- ϵ is some small value (default 0.00001) added to the variance in case the variance is 0 so we avoid a division by zero error
- γ and β are learnable parameter vectors with the same size as the input size, γ represents weights and β represents bias

2.2.2 Learning Rate Scheduling

Large learning rates lead to problems of bouncing around minima and divergence whereas small learning rates lead to problems of time taken for convergence and being stuck at local minima. Ideally, we want a learning rate that changes at various points during learning to mitigate these issues. Learning rate scheduling does this by reducing the learning rate by a factor of α either every n epochs or when learning stagnates. Specifically, pytorch 's ReduceLROnPlateau reduces the learning rate based on whether or not a metric is improving [9]. The patience parameter specifies how many epochs to wait before reducing the learning rate by a factor specified by the factor parameter.

2.2.3 Adaptive Pooling

In normal max and average pooling, we set the parameters for kernel size and stride. However, in adaptive pooling, we set the desired output size and the corresponding stride and kernel size is set such that after the pooling operation we will have an output of the shape that we specified. Adaptive average pooling uses the average method for the pooling whereas adaptive max pooling uses the max method for the pooling.

2.2.4 Data Augmentation

Sometimes during training, we may find that our model is overfitting on the training data and producing poor results on our test/validation data and this might be largely due to the network not being able to generalize well and/or tending to overfit the training data. Luckily for tasks such as image classification, we can augment our data relatively cheaply through image flipping, cropping, normalizing, etc. This is easily done using pytorch's torchvision. transforms library, specifically with the help of the compose function which simplifies the chaining of various transforms. The training images are augmented as they are loaded with some transformations having probabilities which mitigates overfitting as the network is given more general data per batch. In a way by diversifying the training data, we are providing the network with more data and generally the mode data the better.

3. The Dataset

In train. csv, we see the following structure:

Table 2. Training data layout

image_id	label
xxxxxxxxxx.jpg	Number from 0 to 4 inclusive

Where image_id refers to the corresponding image file in the folder train_images and label refers to a classification from:

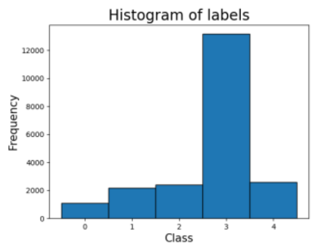
- 0: Cassava Bacterial Blight (CBB)
- 1: Cassava Brown Streak Disease (CBSD)
- 2: Cassava Green Mottle (CGM)
- 3: Cassava Mosaic Disease (CMD)
- 4: Healthy

There are 21,367 training images located in the folder train_images, however only 1 test image located in the folder test_images. A holdout set from the training images will serve as the validation set.

4. Data Exploration

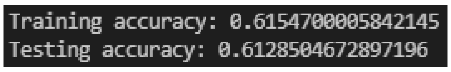
The first order of business is to examine the distribution of labels.

Fig 1.7 Histogram of labels in train. csv (see label_histogram() in plots. py for code)



Clearly, class 3 is the most dominant in the entire dataset. It appears 13158 times which accounts for ≈ 61.50% of the dataset. This means that if we (naively) create a model which just simply predicts the most common class (class 3 in this case), the model will have an accuracy of around 61% (categorization accuracy is the evaluation metric for this Kaggle competition) on the training set. To test this, I created the naïve model and recorded the training and validation set accuracy.

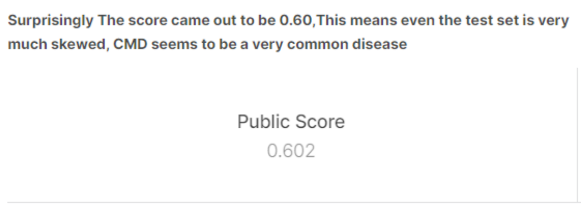
Fig 1.8 Output of naive_model. py



```
Training accuracy: 0.6154700005842145
Testing accuracy: 0.6128504672897196
```

But of course, this is to be expected as a split in the set should also approximately proportionately distribute all classes in both training and validation sets. For this competition, the actual test set is private, meaning it's only available for testing a model after submission. Kaggle user mohneesh7 in this notebook submitted the naïve model and found that the test set is also skewed in favour of class 3 [4].

Fig 1.9 Results of submitting the naïve model to the Kaggle competition [4]

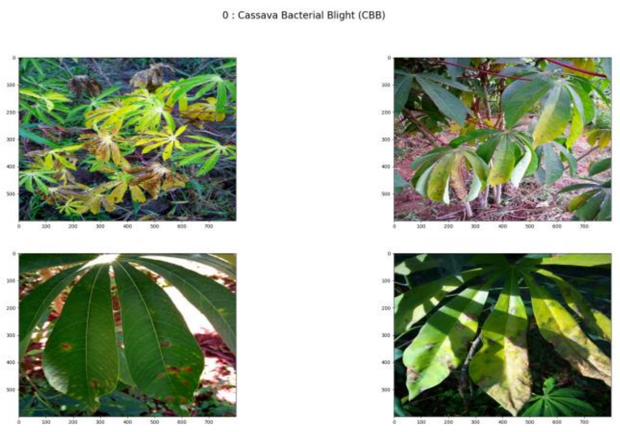


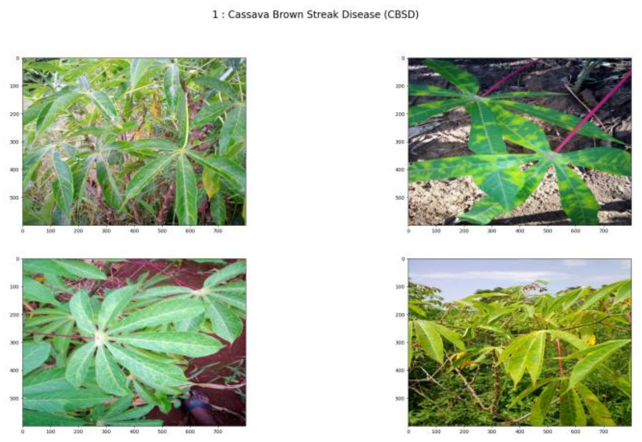
Surprisingly The score came out to be 0.60,This means even the test set is very much skewed, CMD seems to be a very common disease

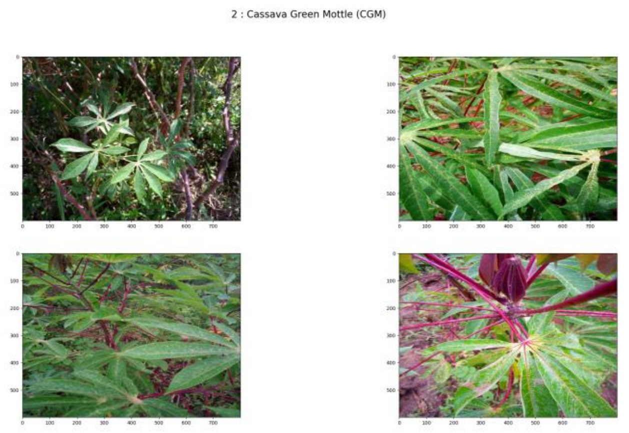
Public Score
0.602

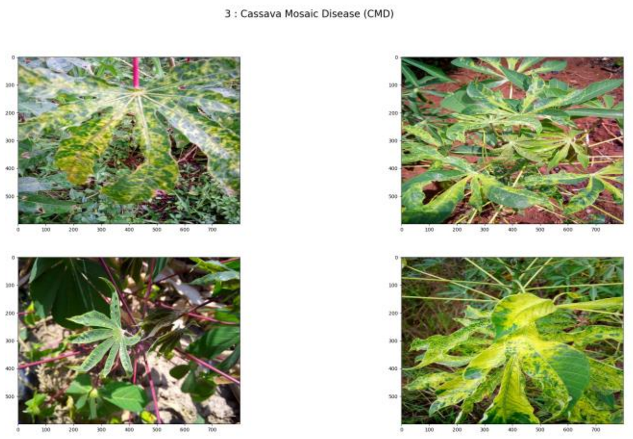
Having now established the class imbalance observation, I will now look at specific examples from each class.

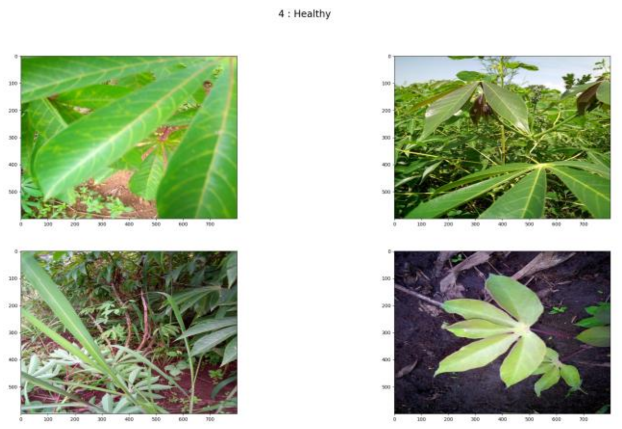
Fig 2.0 Examples of images of classes (see plot_class in plots. py for code)











Empirically it can be seen that each disease is characteritsed by features that can be picked up visually. For class 0: the dried out, brown spots on the cassava leaves. For class 1: the lighter green splotches on the leaves. For class 2: light series of dots on the leaves. For class 3: as the name suggests, a mosaic of lighter splotches and dots on the leaves. For class 4: largely green and devoid of any blemishes such as light splotches and dots.

The above characteristics are all finer features within the images. Relevant image processing techniques that can highlight such features can be applied and then fed into a classifier such as SVM for desirable results, however the SVM without such aid will most likely struggle to identify these finer details. Therefore, these observations further motivate the usage of convolutional neural networks (CNNs) for this classification task.

5. Implementations

5.1 Convolutional Neural Network Implementations

The code for this section is split up in the files: $\mathrm{dataloader.py}$, $\mathrm{epoch\_train\_val.py}$, $\mathrm{main\_dl.py}$, $\mathrm{models.py}$, $\mathrm{utils.py}$

5.1.1 The Models

Let

$$\mathrm{Layers(a,b,c,d)} = \left[\mathrm{Layer}_1^{(a)}\right] \rightarrow \left[\mathrm{Layer}_2^{(b)}\right] \rightarrow \left[\mathrm{Layer}_3^{(c)}\right] \rightarrow \left[\mathrm{Layer}_4^{(d)}\right] \rightarrow$$

And recall that:

$$\mathrm{Layer}^{(n)} = [\mathrm{Block}_1|\mathrm{BigBlock}_1] \rightarrow \cdots \rightarrow [\mathrm{Block}_n|\mathrm{BigBlock}_n] \rightarrow$$

$$\mathrm{Block} = (\mathbf{x})\ \mathrm{Conv} \rightarrow \mathrm{Batch\ Norm} \xrightarrow{\mathrm{ReLU}} \mathrm{Conv} \rightarrow \mathrm{Batch\ Norm} \xrightarrow{\mathrm{IDS}} (+\mathbf{x}) \xrightarrow{\mathrm{ReLU}}$$

$$\mathrm{BigBlock} = (\mathbf{x})\ \mathrm{Conv} \rightarrow \mathrm{Batch\ Norm} \xrightarrow{\mathrm{ReLU}} \mathrm{Conv} \rightarrow \mathrm{Batch\ Norm} \xrightarrow{\mathrm{ReLU}} \mathrm{Conv} \rightarrow \mathrm{Batch\ Norm} \xrightarrow{\mathrm{IDS}} (+\mathbf{x}) \xrightarrow{\mathrm{ReLU}}$$

I will be evaluating 4 models on this dataset.

Stacked CNN with the following architecture:

$$\mathrm{Input} \rightarrow [\mathrm{ConvBlock}_1] \rightarrow \cdots \rightarrow [\mathrm{ConvBlock}_5] \rightarrow \mathrm{AAP} \rightarrow [\mathrm{FC\ Layer}] \xrightarrow{\mathrm{Log\ Softmax}} \mathrm{Output}$$

$$\mathrm{ConvBlock} = \mathrm{Conv} \xrightarrow{\mathrm{ReLU}} \mathrm{Max\ Pool} \rightarrow$$

$$\mathrm{AAP} = \mathrm{Adaptive\ Average\ Pool}$$

$$\mathrm{FC\ Layer} = \mathrm{Dropout} \rightarrow \mathrm{Linear} \xrightarrow{\mathrm{ReLU}} \mathrm{Dropout} \rightarrow \mathrm{Linear} \xrightarrow{\mathrm{ReLU}}$$

ResNet18 which has the following architecture:

$$\mathrm{Input} \rightarrow \mathrm{Conv} \rightarrow \mathrm{Batch\ Norm} \xrightarrow{\mathrm{ReLU}} \mathrm{Max\ Pool} \rightarrow [\mathrm{Layers}(2,2,2,2)]_{\mathrm{Block}} \rightarrow \mathrm{AAP} \rightarrow \mathrm{Linear} \rightarrow \mathrm{Output}$$

ResNet34 which has the following architecture:

$$\mathrm{Input} \rightarrow \mathrm{Conv} \rightarrow \mathrm{Batch\ Norm} \xrightarrow{\mathrm{ReLU}} \mathrm{Max\ Pool} \rightarrow [\mathrm{Layers}(3,4,6,3)]_{\mathrm{Block}} \rightarrow \mathrm{AAP} \rightarrow \mathrm{Linear} \rightarrow \mathrm{Output}$$

ResNet50 which has the following architecture:

$$\mathrm{Input} \rightarrow \mathrm{Conv} \rightarrow \mathrm{Batch\ Norm} \xrightarrow{\mathrm{ReLU}} \mathrm{Max\ Pool} \rightarrow [\mathrm{Layers}(3,4,6,3)]_{\mathrm{BigBlock}} \rightarrow \mathrm{AAP} \rightarrow \mathrm{Linear} \rightarrow \mathrm{Output}$$

Note: $[\mathrm{Layers(a,b,c,d)}]_{\mathrm{B}}$ means that each layer is using block type B which is either Block or $\mathrm{BigBlock}$.

For the Stacked CNN:

Table 4. Stacked CNN parameter details, the rest of the c_i and c_o depend on the first c_o (one of 32, 64 and 80)

	c_i	c_o	K_s	s	p	m_k	m_s
$\mathrm{ConvLayer}_1$	3	32|64|80	11	4	2	3	2
$\mathrm{ConvLayer}_2$	32|64|80	96|192|240	5	1	2	3	2
$\mathrm{ConvLayer}_3$	96|192|240	192|384|480	3	1	1	N/A	N/A
$\mathrm{ConvLayer}_4$	192|384|480	128|256|320	3	1	1	N/A	N/A
$\mathrm{ConvLayer}_5$	128|256|320	128|256|320	3	1	1	3	2

Where: c_i = number of input channels, c_o = number of output channels, K_s = kernel size, s = stride, p = padding, m_k = max pool kernel size, m_s = max pool stride.

Adaptive average pool output size is set to $(6,6)$. All dropout layers had a probability of 0.25.

The Stacked CNN consisted of 3 linear layers which have the following structure:

First layer:

- $128 \times 6 \times 6 = 4608$ inputs, 2048 outputs for first $c_o = 32$
- $256 \times 6 \times 6 = 9216$ inputs, 4096 outputs for first $c_o = 64$
- $320 \times 6 \times 6 = 11520$ inputs, 8192 outputs for first $c_o = 80$

Second layer:

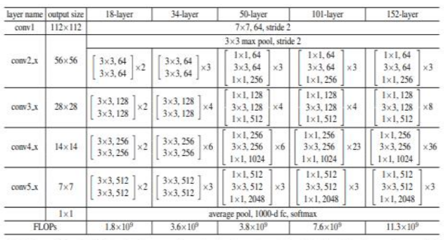
- 2048 inputs, 2048 outputs for first $c_o = 32$
- 4096 inputs, 4096 outputs for first $c_o = 64$
- 8192 inputs, 8192 outputs for first $c_o = 80$

Third layer:

- 2048 inputs, 5 outputs for first $c_o = 32$
- 4096 inputs, 5 outputs for first $c_o = 64$
- 8192 inputs, 5 outputs for first $c_o = 80$

For ResNet:

Fig 2.1 Table summarizing structure of different ResNet architectures [7], only concerned about 18-layer, 34-layer, and 50-layer.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Table 1. Architectures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of blocks stacked. Down-sampling is performed by conv3_1, conv4_1, and conv5_1 with a stride of 2.

5.1.2 Model Training and Results

To test each model to ensure whether it would be appropriate for extensive training, the dataset was reduced to 5000 randomly sampled data points, this was reproducible for training all the models due to setting the random_state parameter. The resulting dataset was split into training and validation sets with a ratio of 0.2 (4000 training, 1000 validation). The optimizer used was stochastic gradient descent (SGD) with an initial learning rate of 0.01 and a momentum of 0.9 for all models. The ReduceLROnPlateu scheduler (refer to section 2.3.2) was used for learning rate scheduling with a patience of 2 epochs and 0.2 was the factor for learning rate reduction. A batch size of 16 was used for all models. All models were trained for 50 epochs. There was data augmentation for the training sets of each model, with the augmentations being: random resized crop of size (384, 384), random horizontal flip with a probability of 0.5, random vertical flip with a probability of 0.5 followed by normalization. The test sets for each model simply had each image resized to (384, 384) and then normalized. The evaluation metric is sklearn 's accuracy_score. The results of this training for each model follow.

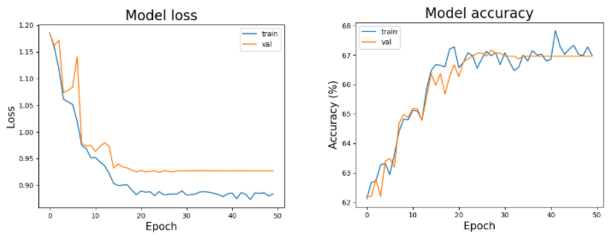
Table 5. Summary of training conditions

Number of training datapoints	4000
Number of validation datapoints	1000
Optimizer	Stochastic Gradient Descent (SGD) Initial learning rate: 0.01 Momentum: 0.9
Learning Rate Scheduler	ReduceLROnPlataeu Patience: 2 epochs Factor: 0.2
Batch Size	16
Number of Epochs	50
Data Augmentations	Random resized crop (384, 384) Random horizontal flip (p = 0.5) Random vertical flip (p = 0.5) Normalize (μ and σ taken over entire training set)

There were 3 different configurations of the Stacked CNN (refer to table 4): one for each different starting $c_o^{(1)}$ = 32, 64 or 80. The results for each are as follows:

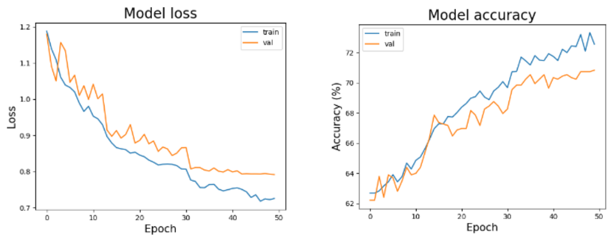
Stacked CNN$_{c_o^{(1)}=32}$

Fig 2.2 Stacked CNN$_{c_o^{(1)}=32}$ model accuracy and loss



Stacked CNN$_{c_o^{(1)}=64}$

Fig 2.3 Stacked CNN$_{c_o^{(1)}=64}$ model accuracy and loss



Stacked CNN$_{c_o^{(1)}=80}$

Fig 2.4 Stacked CNN$_{c_o^{(1)}=80}$ model accuracy and loss

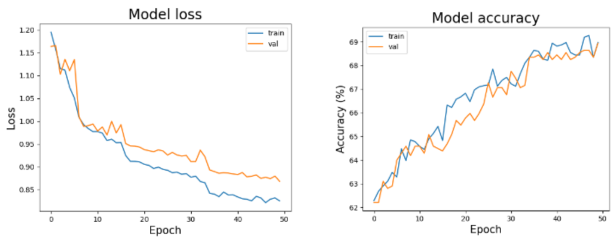


Table 6. Summarized best loss and accuracies for Stacked CNN models

	Best validation loss (2 d.p.)	Best validation accuracy (nearest %)
Stacked CNN$_{c_o^{(1)}=32}$ (1)	0.94	67%
Stacked CNN$_{c_o^{(1)}=64}$ (2)	0.80	71%

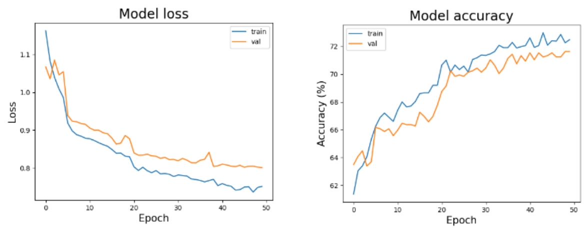
Stacked CNN$_{c_o^{(1)}=80}$(3)	0.88	69%

Model (1) had the lowest validation set loss and accuracy and as seen in Fig 2.2, the validation set loss and accuracy started stagnating at around the 20th epoch. Model (2) had the best validation set loss and accuracy and suffered a similar stagnation that started at around the 40th epoch (refer to fig Fig 2.3). Finally, model (3) had the second-best validation set loss and accuracy but didn't have any evident stagnations (refer to Fig 2.4) as seen for the other 2 models. This means that if model (3) was trained for more epochs then it may have yielded better results than the other models, but at the cost of computation time. With that in mind, Stacked CNN$_{c_o^{(1)}=64}$ will be chosen for further extensive training as it is a suitable trade-off between accuracy and computation time.

After finding out the best Stacked CNN to use, the ResNet[18|34|50] models were trained under the exact same conditions (refer to table 5). The results follow.

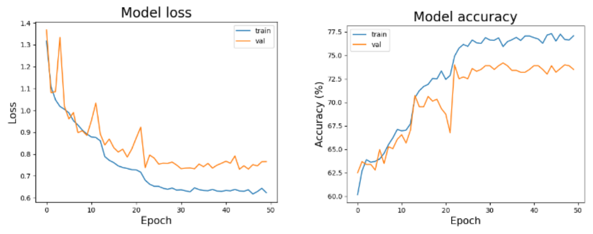
ResNet18

Fig 2.5 ResNet18 model accuracy and loss



ResNet34

Fig 2.6 ResNet34 model accuracy and loss



ResNet50

Fig 2.7 ResNet50 model accuracy and loss

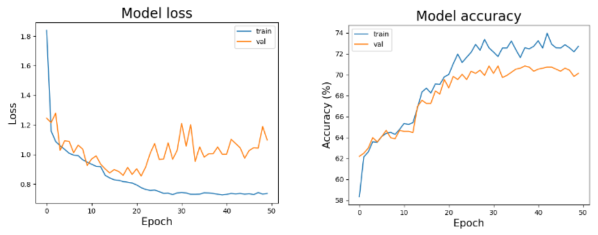


Table 7. Summarized best loss and accuracies for ResNet models

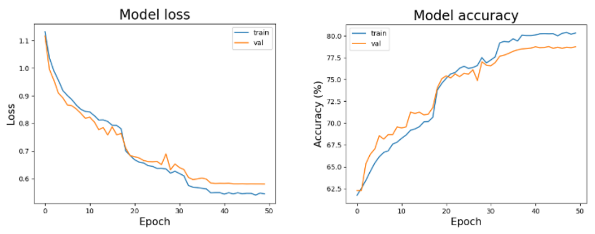
	Best validation loss (2 d.p.)	Best validation accuracy (nearest %)
ResNet18	0.80	72%
ResNet34	0.75	73%
ResNet50	0.88	70%

ResNet18 and ResNet34 showed similar results in terms of the validation accuracy, with ResNet34 having the highest validation set accuracy and the lowest loss, therefore ResNet34 seems more promising in terms of training on more data than ResNet18 . ResNet50 had the highest validation loss and lowest validation accuracy, it may have been improved on using more training data and more epochs but once again due to computational time constraints it is not as feasible as just using ResNet18 or ResNet34 on the full dataset.

With that being said, the models Stacked CNN$_{c_o^{(1)}=64}$, ResNet18 and ResNet34 will be trained on the full dataset. The training conditions are the same as in table 5 except the number of training datapoints is now **17117** and the number of validation datapoints is now **4280**. The results for this training follow.

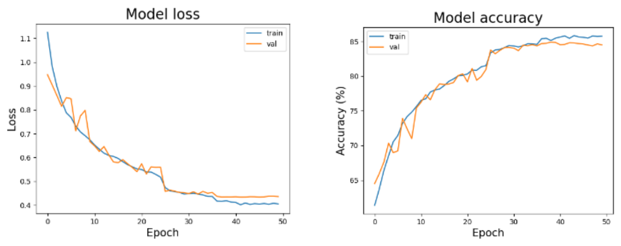
Stacked CNN$_{c_o^{(1)}=64}$

Fig 2.8 Stacked Cnn$_{c_o^{(1)}=64}$ model accuracy and loss on full dataset



ResNet18

Fig 2.9 ResNet18 model accuracy and loss on full dataset



ResNet34

Fig 3.0 ResNet34 model accuracy and loss on full dataset

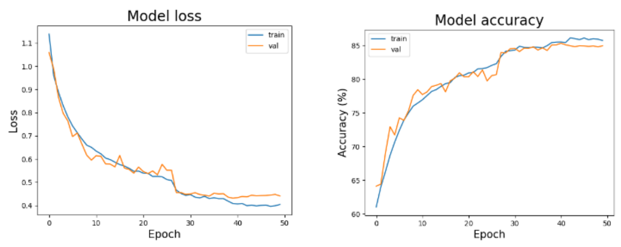
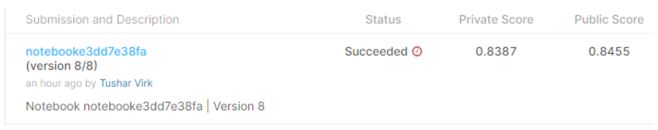


Table 8. Summarized best loss and accuracies for models trained on the full dataset

	Best validation loss (2 d.p.)	Best validation accuracy (nearest %)
Stacked CNN$_{c_o^{(1)}=64}$	0.60	78%
ResNet18	0.43	83%
ResNet34	0.42	85%

ResNet34 displayed the best validation loss and accuracy which was closely followed by ResNet18 . Stacked CNN$_{c_o^{(1)}=64}$ had by far the worse validation loss and accuracy. In all models, loss and accuracy started stagnating at around epoch 30 (refer to Figs 2.8, 2.9, 3.0), with Stacked CNN$_{c_o^{(1)}=64}$ displaying the highest stagnation in terms of test accuracy and ResNet34 displaying the lowest. How closely the validation line plot fits the training line plot in Fig 3.0 is noteworthy because it means that specifically for ResNet34, the model is generalizing well to the training data. With all that factored in and since there isn't too significant a difference between computation time for ResNet18 and ResNet34, ResNet34 will be a suitable choice for our final model. This model was submitted to the competition on Kaggle to evaluate its performance on the actual hidden test set. The model achieved a public score of 0.8455 and a private score of 0.8387. The submission notebook can be found here. These scores are like the scores on the validation set, meaning the validation set can be considered to be a useful evaluation of our models.

Fig 3.1 Kaggle private and public score for ResNet34 model



Submission and Description	Status	Private Score	Public Score
notebooke3dd7e38fa (version 8/8) an hour ago by Tushar Virk	Succeeded	0.8387	0.8455

Notebook notebooke3dd7e38fa | Version 8

I will end this section by looking at the feature map plots (i.e. result of applying the learned filters) for the following image (class label 0).

Fig 3.2 Class label 0 image

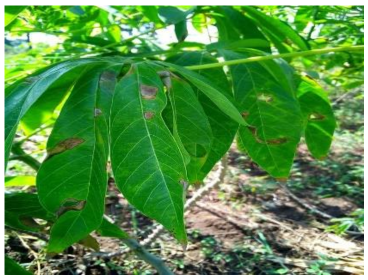


Fig 3.3 Subset of feature maps of the 1st convolutional layer

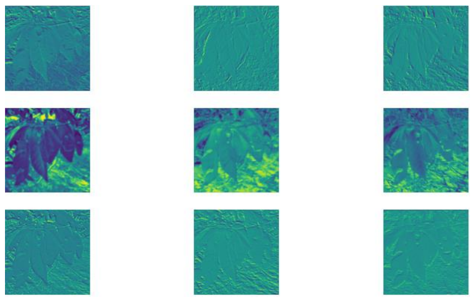


Fig 3.4 Subset of feature maps of the 17th convolutional layer

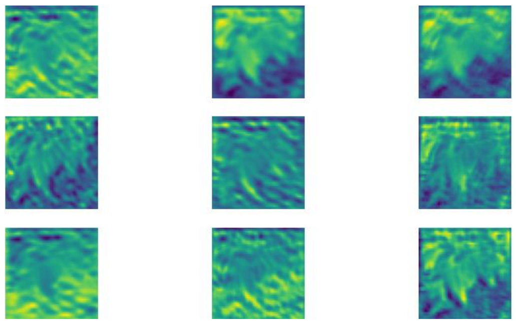
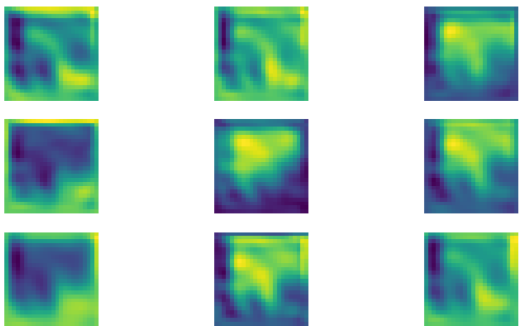


Fig 3.5 Subset of feature maps of the last convolutional layer



From figures 3.3, 3.4 and 3.5 it is clear that the network learns high level features such as edges in the earlier layers and then the deeper we go, the finer the details that the network learns get. This justifies the decision to use CNNs for this classification task as our final model was able to (to a decent degree) extract the fine details that categorize each class (refer to section 4. Data Exploration) in the feature maps and consequently achieve a reasonable validation score of 0.85, private test score of 0.8387 and public test score of 0.8455.

6. Conclusion and Future Work

6.1 Conclusion

To conclude, convolutional neural networks (CNNs) proved to be a very suitable deep learning algorithm for this image classification task. Various models at different stages were evaluated and eliminated with the goal to end the training process with the best possible model. The $\mathrm{StackedCNN}$ model was evaluated for different choices of $c_o^{(1)} = 32,64,80$ on a subset of size 5000 of the entire dataset and it was found that $\mathrm{StackedCNN}_{c_o^{(1)}}$ produced the most desirable results. $\mathrm{StackedCNN}_{c_o^{(1)}}, \mathrm{ResNet18}, \mathrm{ResNet34}$ and $\mathrm{ResNet50}$ were then evaluated on the same data subset and $\mathrm{ResNet34}$ proved to have the best computation time – accuracy trade-off. Finally, $\mathrm{ResNet34}$ was evaluated on the entire dataset and produced a final validation accuracy of 85%. The final model can to a reasonable extent detect diseases given an image of a cassava leaf.

6.2 Future Work

Some considerations for future work:

Neural transfer learning. All the models in this project were trained from ‘scratch’ (random initial weights). A potential way to produce a better model in the future would be to use pre-trained models on large datasets such as $\mathrm{ImageNet}$.

Hyperparameter tuning. Out time constraints, there was no structured hyperparameter tuning performed during the project. It will be useful in the future to tune hyperparameters such as learning rate, momentum, batch size, etc. and document the results.

Network ensembles. All the models in this project were singular models producing an output which was deemed final. Network ensembles could have been utilized to reduce variance and improve accuracy. This could work well with the other methods such as transfer learning and eventually produce a high-quality model.

7. References

[1] Dresden, D., 2021. Cassava: Benefits, toxicity, and how to prepare. [online] medicalnewstoday.com. Available at: https://www.medicalnewstoday.com/articles/323756#what-is-it.

[2] Kaggle.com. 2021. Cassava Leaf Disease Classification | Kaggle. [online] Available at: https://www.kaggle.com/c/cassava-leaf-disease-classification/overview.

[3] Lindsay, G. (2020). Convolutional Neural Networks as a Model of the Visual System: Past, Present, and Future. Journal of Cognitive Neuroscience, pp.1–15.

[4] kaggle.com. (n.d.). Cassava 2020(Case study)[Full Progress][keras+GPU]. [online] Available at: https://www.kaggle.com/mohneesh7/cassava-2020-case-study-full-progress-keras-gpu.

[5] Saha, S. (2018). A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. [online] Towards Data Science. Available at: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[6] image-net.org. (n.d.). ImageNet. [online] Available at: https://image-net.org/.

[7] He, K., Zhang, X., Ren, S. and Sun, J. (2015). Deep Residual Learning for Image Recognition. [online] . Available at: https://arxiv.org/pdf/1512.03385.pdf.

[8] pytorch.org. (n.d.). BatchNorm2d — PyTorch 1.9.0 documentation. [online] Available at: https://pytorch.org/docs/stable/generated/torch.nn.BatchNorm2d.html.

[9] pytorch.org. (n.d.). ReduceLROnPlateau — PyTorch 1.9.0 documentation. [online] Available at: https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html.