Division of Computing, Engineering and Mathematical Sciences
Department of Computing, University of Kent *in Canterbury*

Plant 'Cassava' Disease Classification

By

Lukasz Tomaszewski, Darren Booker, Jamie Lennard and Rowan Glancey

lrgt2@kent.ac.uk, dmb48@kent.ac.uk,
jl422@kent.ac.uk and rh601@kent.ac.uk

Module CO880: Project & Dissertation

LaTeX Word Count: 4,360

April 2022

Contents

Abstract

To further explore artificial intelligent systems within society, a cognitive neural network model that detect a disease in the cassava plant is taken and improved. Initial issues arise in the model's runtime and GPU requirement allowing only a small number of epochs to happen, where theoretically, the more epoch the better the test accuracy. Outsourcing the GPU via a python web server on Google Collab and the runtime issue resolved , it was found that the system could have the potentially to improve the test accuracy up 67% given certain criteria met through training layer independently from each other. Through testing, it was proven that the more epoch ran, the better the test accuracy however the limit was beyond the GPU restrictions but could be seen in further development. The hindrances did not allow for the projects goals to be completed however sufficient improvement was made unto the existing model and is set for further development.

Report Link: https://git.cs.kent.ac.uk/jl742/comp8260-group-project/-/tree/jl742-2

1 Introduction

Artificial intelligence is used in various industrial and research fields to assist humanity in data analysis, medical care and recovery and more. To further enhance and explore the implementation of artificial intelligent systems detecting a disease within the cassava plant. In the project a collection of image of the plants will vary from healthy, slightly infected, fully infected and dead, "This group will endeavour to improve an existing classifier on a current data set. This groups goals are to experiment with the PCA, Decision Trees, Random Forests and Neural Networks on a data set to classify images displaying different stages of the 'Cassava' disease on plants. The data set [1] selected consists of thousands of images, we initial view this as an opportunity to compare the run time and accuracy of the machine learning script on different number of images supplied to the script, one could logically state that the more images analysed then the longer the run time and more accurate it will be to distinguish the identifier.

However what interests this group it how much longer will the run time be and how much will the accuracy change, once this analysis is complete, we can explore using different filters, change the script and change the image format to shorten the run time and improve accuracy. Upon initial review of the requirements, the data set is 12gb (Mostly copies of images in different formats) but only consists of 3.14gb of usable images, the Cassava data set [1] and script will be written in python which all members have and will be linked to commit to GitLab regularly. No other significant requirements are visible at this stage. The group chosen this data set as it's feasible that it can be manipulated and rewritten, we all are comfortable and believe the project goals are achievable with our current understanding.

The initial plan is to divide the workload whilst all committing to the project on GitLab. The report, research, presentation and analysis will be given to each member as their primary contribution to the project. Each group member will explore different number of images supplied to the final script to which will be analysed later via a graph displaying run time, accuracy against number of supplied images. The initial research is on-going currently as we begin to write the ML script, to which we will run a proof of concept and

run 5k, 10k, 20k images and compare the run time, accuracy. In the second sprint we refine the Python script repeat the proof of concept which will give us a good comparison to how we have improved the Python script." [**?**]

2 Methodology

In the Cassava_autoencoder_pretraining project [**?**] The merged_train.csv file is the data set index file which is stored in the current working directory (CWD). It holds 27053 rows of sample image filename ('image_id') and an integer representing the sample category ('labels'). The train_images folder holds the 27053 images in jpeg format. The data set used was attached to this project was Cassava plant disease classification, it was downloaded as an archive of images with a file that acted as a dictionary for the classification based on the file names. The first thing we had to do was have a pre-processing stage to get the images into a usable format that we could begin the machine learning on. The first thing to do was extraction which is easy enough in python using the 'zipfile' library. Once the images are extracted we then move the images to locations based on what they represent, in the beginning we tried having it so that we split the label array based on the images then split it all based on training and testing however this became too much complexity for what it was worth. This was then changed to move the images to a train and testing directory based on a 50/50 split of each of the classes, each of the images were put into a sub-directory in the training/testing parent based on the class they represent, so the format was as follows:

Cassava/train/CBB
/CBSD... etc
/test/CBB... etc

This format allowed the data to be easily read into keras using the pre-processing function 'image_dataset_from_directory' which could infer the labels based on the folder name that the image was in. This also allowed us to easily split the data into a validation set for the training which became useful further on when creating models. This function also allowed for changing the image size, this was reduced as the original images were very large and computationally expensive.

2.1 Cognitive Neural Network

All python settings for the model structure and execution are in model_settings.py. All python functions for manipulating the model are in cnn_func.py.
1. Convolution of image data

The inputs are applied to a Conv2d layer then a pooling layer. There can be up to three of these combinations. If any of FILTERS_1, FILTERS_2, FILTERS_3 are greater than 0, the corresponding Conv2d layer will be created. Conv2d settings: number of filters, activation function, kernel size and stride, pooling settings: pool size and stride.

2. Neural Network Deep Learning

The convolution outputs is then applied to the neural network hidden layers. The hidden layers consist of up to three combinations of a dropout layer then a dense layer. The DENSE_KERNEL_INITIALIZER is the kernel_initializer parameter for the Dense layers. If any of DROPOUT_RATE_1, DROPOUT_RATE_2, DROPOUT_RATE_3 are greater than 0, the corresponding Conv2d layer will be created.

Dropout settings: the dropout rate
If any of DENSE_OUT_1, DENSE_OUT_2, DENSE_OUT_3 are greater than 0, the corresponding Conv2d layer will be created.

Dense settings: number of units (outputs) and activation function. The hidden layers can be added at any stage of training. So, convolution can be jump-started before applying deep learning. Additional hidden layers could also be added at any time. If INITIAL_DENSE_LAYERS is False then the hidden dropout and dense layers are not initially created.

3. Output probabilities for five categories.

The final layer is a dense layer that produces one output for each category. The output layer settings are the number of classes (OUT) and the activation function (OUT_ACTIVATION).

Model Compilation:
The OPTIMIZER_NAME is the name of the compilation optimizer parameter.
The MODEL_OPTIMIZER is the instance for the compilation optimizer parameter.
MODEL_LOSS is the compilation loss parameter.
MODEL_METRICS is the compilation metrics parameter.

Model Storage:
All models are saved to the MODEL_DATA_DIR using the MODEL_DATA_FILENAME. The MODEL_DATA_FILENAME is 'OPTIMIZER_NAME.lower()_PREPARED_IMAGES_FOLDER'. The data set index files use a prefix setting. MODEL_TRAIN_PREFIX is 'train_ind' and MODEL_TEST_PREFIX is 'test_ind'.

3 Results & Findings

3.1 CNN Model

When loading sample images for processing, the requirements are usually too high to hold every sample image in a training or test set. So, only a subset of the sample images is loaded at any time. The Tensorflow (keras) image_dataset_from_directory function could be used. However, the resource demand for modest computers is high. So, first compare with the image_array method. A subset of the sample images is loaded into an array using a range of the index data frame rows. Over time, adaptations to how the image samples are fitted gave greater control over data variation. Originally lading batches into an array, then using train and test directories. Finally the test and train index DataFrames were adapted to load the images which eliminates the need for copying the image data. Many

model configurations were explored using various optimizers, such as Adam, Adamax, SGD, Adadelta.

The number of fitted samples before the weights are updated is given with the batch_size parameter of the fit function. It seemed reasonable that increasing the batch_size should speed-up processing at the cost of some accuracy. Training was performed on a very simple model, passing 1000 images for each fitting (ARRAY_IMAGES_TRAIN) with the batch_size parameter set to 100 (FIT_BATCH_SIZE). Throughout training, the computer was unusable and Tensorflow displayed many warning messages. The training took over half a day! However, a simple adjustment to the batch_size reduced runtime to just 1.5 hours and improved accuracy, while eliminating many of the warning messages. The run-time was reduced to just 1.5 hours.

The best (latest) model uses train and test index files with original sized images so does not duplicate the data set. The fraction of the data set used for training is 0.85. The sample images are loaded with width of (280, 210) and 3 normalized channels for pixel data. They randomly flip horizontally, have a random rotation range of 40, random width and height of 0.2. Fitting has a batch size of 10, a learning rate between 0.1 and 0.01, and 5 epochs executed twice at a time. The kernel initializer is GlorotNormal for all layers using the Adadelta optimizer.

All layers use the relu activation function except the output layer which uses softmax. All convolution layers use 3x3 kernels with 1x1 strides and are followed by a pooling layer with 2x2 pools and strides. The three convolution layers have 64, 128, and 256 filters respectively.All hidden dense layers (not the output layer) have a preceding dropout layer to ovoid over-fitting. The three hidden dense layers have 396, 792, and 198 outputs with pre-dropout of 0.2, 0.1, and 0.1 respectively. The first 50 epochs (5 iterations) has a learning rate of 0.1 and does not have any hidden layers. Training takes about 2h 43m 28.57s for 10 epochs. The next 50 epochs (5 iterations) have a learning rate of 0.01. Training takes about 2h 43m 28.57s for 10 epochs. The next 50 epochs add the hidden layers and trains only the dense layers at a learning rate of 0.01. Training takes about 1h 36m 1.58s for 10 epochs. Evaluation takes about 5m 19.17s for the training set and 55.67s for the testing set.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 208, 278, 64)	1792
max_pooling2d (MaxPooling2D)	(None, 104, 139, 64)	0
conv2d_1 (Conv2D)	(None, 102, 137, 128)	73856
max_pooling2d_1 (MaxPooling 2D)	(None, 51, 68, 128)	0
conv2d_2 (Conv2D)	(None, 49, 66, 256)	295168
max_pooling2d_2 (MaxPooling 2D)	(None, 24, 33, 256)	0
flatten (Flatten)	(None, 202752)	0
dropout (Dropout)	(None, 202752)	0
dense (Dense)	(None, 396)	80290188
dropout_1 (Dropout)	(None, 396)	0
dense_1 (Dense)	(None, 792)	314424
dropout_2 (Dropout)	(None, 792)	0
dense_2 (Dense)	(None, 198)	157014
dense_3 (Dense)	(None, 5)	995

Total params: 81,133,437
Trainable params: 80,762,621
Non-trainable params: 370,816

Best Results

After 120 epochs:

Train loss: 0.9759	**Train acc: 0.6389**
Test loss: 1.0920	**Test acc: 0.6077**
diff 0.1161	**-0.0312**

Figure 1: Model information of the most trained and high accuracy result of the tests.

3.2 Layers & Filters

In order to attempt to optimize the structure and hyper-parameters of the model I took a reduced data set to run short tests on with few epochs. This reduced data set consisted of 400 of each class from the 1500 in the data set and as such would have a very limited ceiling for accuracy on the test data but by comparing these results for test accuracy it was hoped some information could be gleaned about how to structure the model and it's hyper-parameters for use on a larger data et. The first Parameter to decide on is the batch size, as a reasonable trade-off between accuracy and speed needs to be found to test the other parameters in a timely manner. The "starting point" model consisted of a sequential model with a re-scaling layer first to change the RGB values from -256 to 0-1

to work better with the model, then a convolutional layer with 32 filters, a pooling layer, a second convolutional layer with 64 filters, a second pooling layer, a flattening layer and then 2 dense layers.

Both pooling layers used 2x2 pooling with a stride of 2 and both convolutional layers used the relu activation function and a stride of 1. The first Dense layer used the "relu" activation function and the second the "softmax" function. The Loss function used was sparse categorical crossentropy with the adam optimizer. Note that all these results show the model overfitting within the 5 epochs. On the full data set, it is likely that more than 5 epochs would be required to achieve good accuracy with the sufficient data available, but as 5 epochs are sufficient for the validation accuracy to converge to an approximate point, it serves well enough for this purpose. 16 was chosen as a starting point for the batch testing as the training:testing split is 80:20 so the training data is 1600 images, a simple 100 batches per epoch. From the data, it is evident that smaller batch sizes than 16 significantly slow down processing but any positive change to the validation (or training) accuracy is minimal.

Larger batch size values result in a minor speed increase and a minor accuracy decrease, but it is predicted that this minor decrease is only due to the small size of data used and the negative effect on accuracy would be amplified by using this larger batch size on the full data (as evidenced by the lower training accuracy compared to smaller batch sizes), as a result a batch size of 16 was chosen and used for future tests. All of these tests also show the expect result that validation accuracy is never very high, with 20% being as good as guessing, the model never progresses past approximately 30%. This is can be attributed to a lack of optimisation in part but also the simple lack of data. With this value found, I then attempted to optimize the parameters of the layers, first by changing the number of filters used by the convolutional layers.

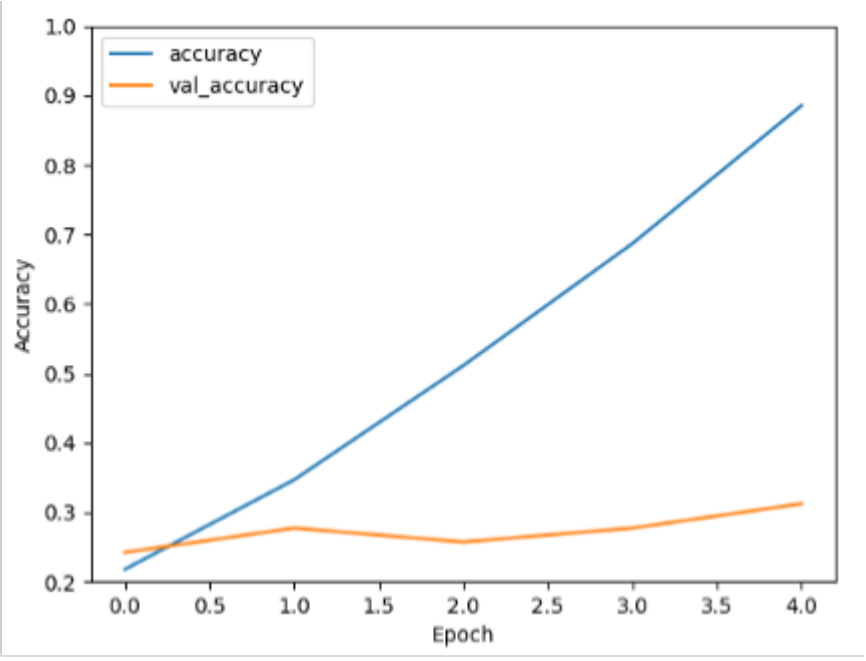

Figure 2: Batch Size =32, taking 31 seconds per epoch.

fig. 9 shows with 32,64 (for the first and second layers respectively) fig. 10 64,128 and fig. 11 16,32. While doubling the filters increased the training accuracy, it had no positive effect on the validation accuracy and made the model take far longer, so is not useful.While halving the filters had the expected effect of decreasing time per epoch, it also had the unexpected effect of increasing validation accuracy significantly (baring in mind that due to limited data to achieve speed in testing there is a cap on how much accuracy can be achieved with any model) with a peak (though not end due to overfitting) accuracy on 39.75%.

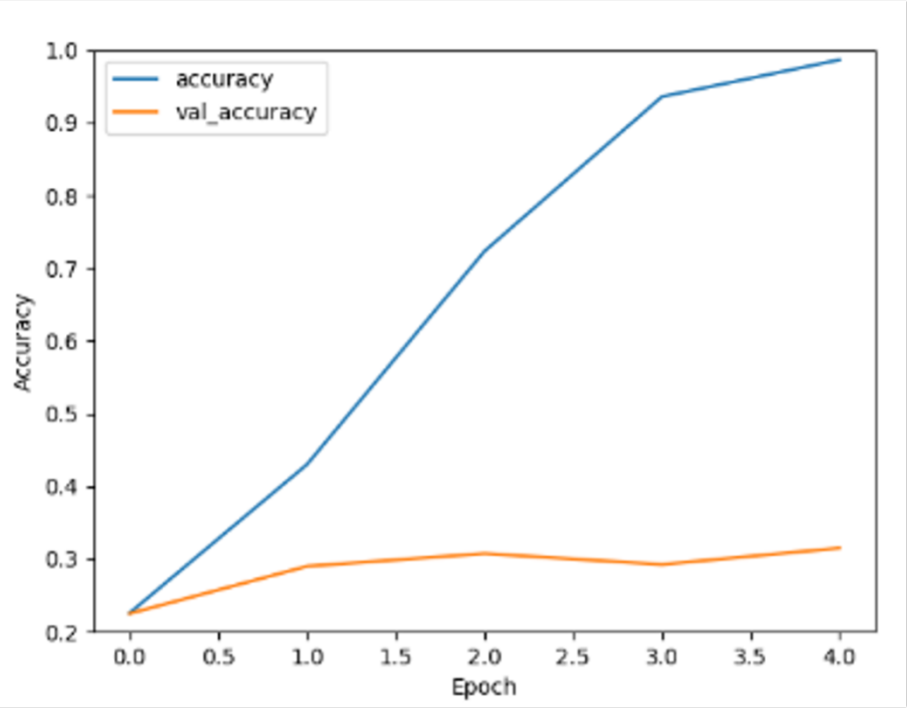

Figure 3: 16,64 Filters. Taking 25 seconds per epoch.

A further decrease to 12/24 Filters however did not result in more accuracy to I then removed to changing the number of filters individually rather than both at once from the 16/32 that had given the best result so far. 16/64, 16/48 and 16/16 all proved not to increase validation accuracy however, with peaks of 31.5% ,35.75% and 32.75% none beat 16/32. This configuration was then tested on the full data set for 5 epochs to get some approximation of it's actual usefulness. This still results in low validation accuracy (34.06%) so I attempted to modify the filters on the full data set as random variance may of accounted for "improvements" on the reduced data set, getting better results with 12/24 though also getting a bizarre result where the validation accuracy starts at it's peak and only decreases with further epochs as the training accuracy increases.

16/32 Filters with No Rescaling

Epochs	1	5
First Training Accuracy	25.95%	45.95%
Last Training Accuracy	25.95%	96.11%
Test Accuracy	24.89%	23.16%
Time Per Epoch (s)	42	42

16/32 Filters with Rescaling

Epochs	1	5
First Training Accuracy	26.08%	35.52%
Last Training Accuracy	26.08%	98.14%
Test Accuracy	31.40%	33.33%
Time Per Epoch (s)	42	42

32/64 Filters with No Rescaling

Epochs	1	5
First Training Accuracy	21.13%	50.57%
Last Training Accuracy	21.13%	95.67%
Test Accuracy	23.55%	22.96%
Time Per Epoch (s)	78	78

32/64 Filters with Rescaling

Epochs	1	5
First Training Accuracy	24.51%	30.64%
Last Training Accuracy	24.51%	79.15%
Test Accuracy	30.70%	33.92%
Time Per Epoch (s)	79	78

While previous tests had all been done with a re-scaling layer, it was worth investigating whether this layer was useful or not, and so the following results were gathered. These results show that adding a re-scaling layer has no noticeable impact on epoch time but seems to increase test accuracy significantly for no downside, so previous assumptions about using the re-scaling layer were correct to do so. The increased first training accuracy on the 5-epoch is because this is done on the model having already one 1 epoch and so this is effectively a 2nd to 6th epoch not a 1st to 5th epoch hence the increased training accuracy initially, not caused by re-scaling.

16/32 Filters with 2x2 filters

Epochs	1	5
First Training Accuracy	26.08%	35.52%
Last Training Accuracy	26.08%	98.14%
Test Accuracy	31.40%	33.33%
Time Per Epoch (s)	42	42

16/32 Filters with 3x3 filters

Epochs	1	5
First Training Accuracy	22.22%	33.74%
Last Training Accuracy	22.22%	98.87%
Test Accuracy	26.33%	35.00%
Time Per Epoch (s)	53	52

32/64 Filters with 2x2 Filters

Epochs	1	5
First Training Accuracy	24.51%	30.64%
Last Training Accuracy	24.51%	79.15%
Test Accuracy	30.70%	33.92%
Time Per Epoch (s)	79	78

32/64 Filters with 3x3 filters

Epochs	1	5
First Training Accuracy	20.8%	30.39%
Last Training Accuracy	20.8%	97.14%
Test Accuracy	20.0%	24.4%
Time Per Epoch (s)	96	92

We then attempted to see if using larger filters of 3x3 instead of 2x2 would help test accuracy giving the results above. This resulted in a noticeable drop in accuracy for the 32/64 filter set and a marginal increase in the 16/32 filter set but this could simply be random variance at work and as it increased epoch time 2x2 filters seem preferable in this current combination of parameters.

16/32 Filters with 256*256 images

Epochs	1	5
First Training Accuracy	26.08%	35.52%
Last Training Accuracy	26.08%	98.14%
Test Accuracy	31.40%	33.33%
Time Per Epoch (s)	42	42

16/32 Filters with 512*512 images

Epochs	1	5
First Training Accuracy	26.98%	46.26%
Last Training Accuracy	26.98%	99.05%
Test Accuracy	33.69%	28.03%
Time Per Epoch (s)	170	170

32/64 Filters with 256*256 images

Epochs	1	5
First Training Accuracy	24.51%	30.64%
Last Training Accuracy	24.51%	79.15%
Test Accuracy	30.70%	33.92%
Time Per Epoch (s)	79	78

32/64 Filters with 512*512 images

Epochs	1	5
First Training Accuracy	27.47%	34.59%
Last Training Accuracy	27.47%	99.02%
Test Accuracy	34.92%	31.68%
Time Per Epoch (s)	312	312

Using the 512x512 Images as inputs drastically increased epoch time and increased both initial and final training accuracy, however test accuracy was actually effected negatively. While the decrease in test accuracy could be due to variance even if this is the case it does not grant any increase in test accuracy for a very large increase in epoch time so the usage of 512x512 images is not useful with this current parameter combination though it could result in increased accuracy under other circumstances.

3.3 Accuracy against Epochs

Once the pre-processing was complete we could move onto modelling, once we got the first basic model working we could begin collecting data but we found that whilst the training accuracy would be very high the testing accuracy would not. The format of these models was two 2D convolution layers with max pooling between, followed by another max pooling layer which was flattened then lead into a large dense layer followed by a final dense layer with 5 classifications with a softmax activation. The default activation for the rest of the model was relu and the pool sizes were (2,2).

This model was tested in various ways, the first of course was adjusting the epochs to see what difference that would make. Moving from 1 epoch to 5 had a slight difference in the testing accuracy but from 5 to 15 had a slight decrease of 0.2%.

Accuracy against number of epochs.			
Epochs	1	5	15
First Training Accuracy	0.21314433217048645	0.21262887120246887	0.8505154848098755
Last Training Accuracy	0.21314433217048645	0.8141752481460571	0.9317010045051575
Testing Accuracy	0.21467182040214539	0.23320463299751282	0.23140282928943634

You can see that more than 1 epoch is clearly favourable however none of the results were ideal with barely a 25% accuracy it may as well be luck that is driving the process. It seems that the model was being heavily overfitted as towards the end of the 15-epoch run all the training accuracies were very close with a range of 2%. This same test of epochs making a difference was done with 2 other models also, one model with double the number of filters in the convolutional layers and one with the sigmoid activation function instead of the relu function. The results were as follows.

Accuracy against number of epochs with double filters.			
Epochs	1	5	15
First Training Accuracy	0.21546392142772675	0.2033505141735077	0.7097938060760498
Last Training Accuracy	0.21546392142772675	0.6847938299179077	0.8432989716529846
Testing Accuracy	0.21441441774368286	0.22651222348213196	0.22857142984867096

Accuracy against number of epochs using the Sigmoid function.			
Epochs	1	5	15
First Training Accuracy	0.20463918149471283	0.20412370562553406	0.20721650123596191
Last Training Accuracy	0.20463918149471283	0.20721650123596191	0.2033505141735077
Testing Accuracy	0.20000000298023224	0.20000000298023224	0.20000000298023224

The double filters did look promising with the gradual increase in training accuracy, I was hoping that the model would not end up over fitting but it still resulted in a 23% accuracy for the testing set. The sigmoid function however seemed to be a complete failure for this set, the accuracy for the training and testing did not change throughout with no difference when increasing the number of epochs either. I would have liked to test the epochs further however with 15 epochs difference granting an increase of 0.0025% the number of epochs would have been massive, especially with a processing time of a minute an epoch. So far, all models have been completed without a validation set, this was changed for the next model as we wanted to test what difference this could make for the training and validation accuracy compared to the testing accuracy at the end.

Epochs	1	5	15
First Training Accuracy	0.2177537977695465	0.5139774680137634	0.975968599319458
Last Training Accuracy	0.2177537977695465	0.9565963745117188	0.9855321049690247
First Validation Accuracy	0.21121923625469208	0.23125357925891876	0.22839152812957764
Last Validation Accuracy	0.21121923625469208	0.24327418208122253	0.23526044189929962
Testing Accuracy	0.2319587618112564	0.20000000298023224	0.223195880651474

Even with validation data there was not a change in the result for the testing. The split for validation was 70/30 here with similar results found at 80/20. An issue that might be causing this could be the split for training and testing data as a whole, so far the split has been 50/50 which could be far too few images for the model to correctly learn. Having more could prevent overfitting and allow for better performance in the testing data set, so the next thing we tried was a 75/25% split for training/testing respectively. This change alone caused negligible changes to the accuracy for training validation and testing however changing the way that the data was input did have a small positive change to the testing accuracy. Normalising the rgb values input into the model by dividing by 255 caused an increase to 30% testing accuracy after 15 epochs, this was by far the highest testing value achieved yet so suggests it is on the right line by doing so. The values for this are as follows.

Epochs	15
First Training Accuracy	0.2457086741924286
Last Training Accuracy	0.9995095729827881
First Validation Accuracy	0.282198041677475
Last Validation Accuracy	0.29421865940093994
Testing Accuracy	0.2994845509529114

After getting a better results from normalising the data we thought to try some different models and to adjust the size of the images from data set, changing the model on its own had lower accuracy than above however by adjusting the image width/height to 224 (as seen elsewhere to be quite common). we also let this model run for 30 epochs as the accuracy's were still gently increasing past 15 and began to plateau around 30 so this is where we took measurements.

Epochs	30
First Training Accuracy	0.20042918622493744
Last Training Accuracy	0.9399141669273376
First Validation Accuracy	0.17768239974975586
Last Validation Accuracy	0.3579399287700653
Testing Accuracy	0.38092783093452454

This yielded the highest testing accuracy so far with 38%. For now we will keep the image size as set here and will continue to adjusting the models to see what impact this will have on the results. The next model we tried was much larger and also included data augmentation which would rotate the images randomly also, this was giving promising results with a peak validation accuracy of 60%, this was on an upward trend however so

the validation could have become stable at 60% should this have been left for longer.

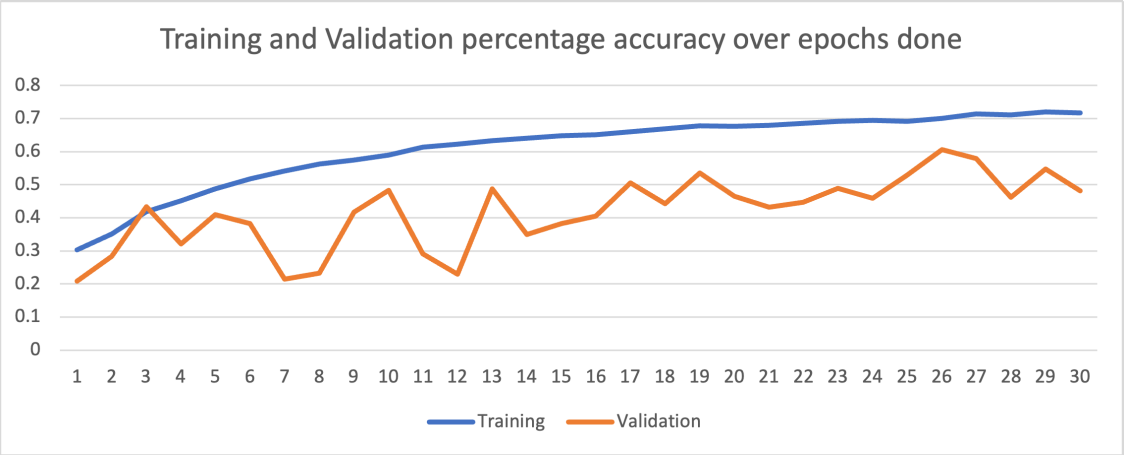

Figure 4: Accuracy of the training and validation data undertaking 30 epoch cycles.

The testing accuracy for this model was 47% which is very close to the validation so if we had stopped the model on that peak it could have performed much better. Since the training accuracy was slowly increasing we felt that there could be more that the model could learn so after continuing the model for between 5-15 epochs at a time it still showed no sign of slowing down with a validation accuracy of 63% and testing of 65% at 85 epochs. Continuing further to 100 continue to show the upward trend of the training accuracy however the validation became more erratic in its changes and on the 100th epoch when testing was done it resulted in an accuracy of 50%, far lower than before.

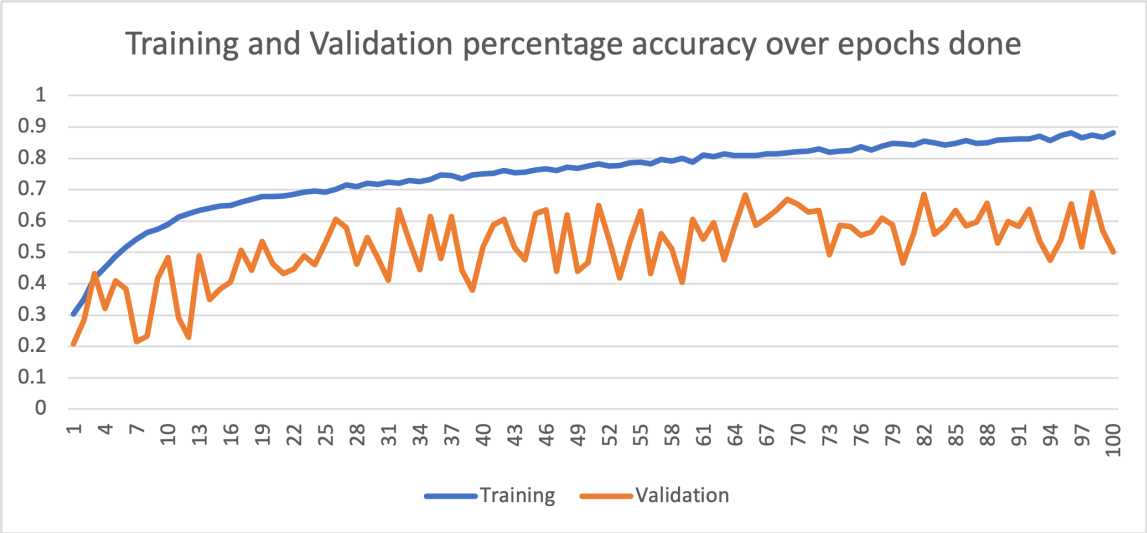

Figure 5: Accuracy of the training and validation data undertaking 100 epoch cycles.

Once again the testing was done on a dip so it is likely by running single epochs on the same model at this point could give a good model with high accuracy for validation. When running the model for single epochs at 101 there was a validation accuracy of 64% and a testing accuracy of 67%, at 102 the validation accuracy went to 65% and the testing

accuracy remained at 67%. 103 and 104 began to dip again giving accuracy's of around 62% however they are all beginning to remain consistently above 60% which was very promising until at 105 when it dipped to 51%. Running continuously now it was clear that it had peaked and was likely to not rise again.

4 Discussion

Our original aims were to explore various classification techniques on a feasible data set were not met, hindered by the model's power requirement. This led to the group looking at ways to further improve the model, keeping in line with the originally proposed cognitive neural network, we looked decreasing the runtime by experimenting with the filters, layers. This was a crux in this group as the power requirements of the GPU were too high, we were able to run 1-10 epochs fine however theoretically the more epochs, the better the test accuracy, a group member had to use and external python web server to run the model efficiently allowing the group to obtain good results. The main issue that arose was that the runtime was too great and when we manage to fix that issue we ran out of time to properly explore the neural network to its fullest capacity. Further improvements would be to explore different classifiers as its apparent that that the CNN classifier has a strict computer requirement. We disregarded the decision trees classifier due to the inaccuracies that is achieves in this type of model and potentially random forests too ass its based off of a true or false algorithm and may disregard key data within the image too soon.

5 Conclusion

As a group we created various models and completed a variety of tests on these models with the main aim of improving accuracy for the testing set. We adjusted the epochs, we changed the shape of the model, the sizes of the images and augmented the images that were being read into the model. The aim was to find the best of all changes to eventually see how they worked together to end up with the best model we could for the Cassava data set, with a peak accuracy of 67% we feel that whilst it could be improved, for the size of the data set and the number of classifiers required of it that this result is very good. The images are complex and very large to start with so for a model with around 100 epochs to get a result like this was a good step forward and shows that perhaps with more changes to the data set, perhaps by flipping images along the vertical, we could have achieved an accuracy closer to 80% with more epochs to train the model. With the Cognitive Neural Network model, training the 2d convolutional layers separately from the dense layer allowed the system to train not just faster but it improved the test accuracy up to 63% which is the highest we've managed to get. The system however requires a lot of power to run the model and we confidently feel that the we have improved this model.

6 Appendix

6.1 A: Accuracy v Epoch Graphs

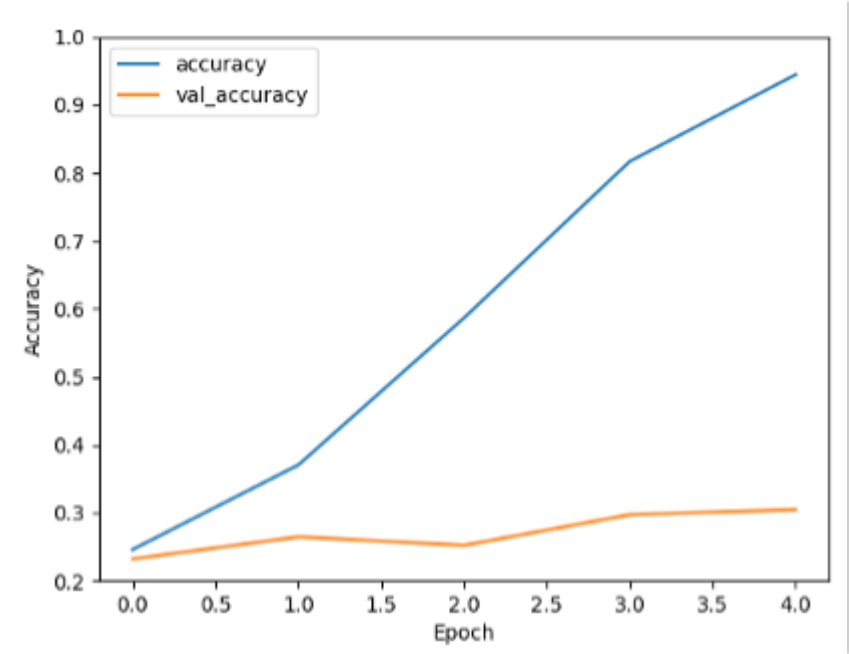

Figure 6: Batch Size = 16, taking 33 seconds per epoch.

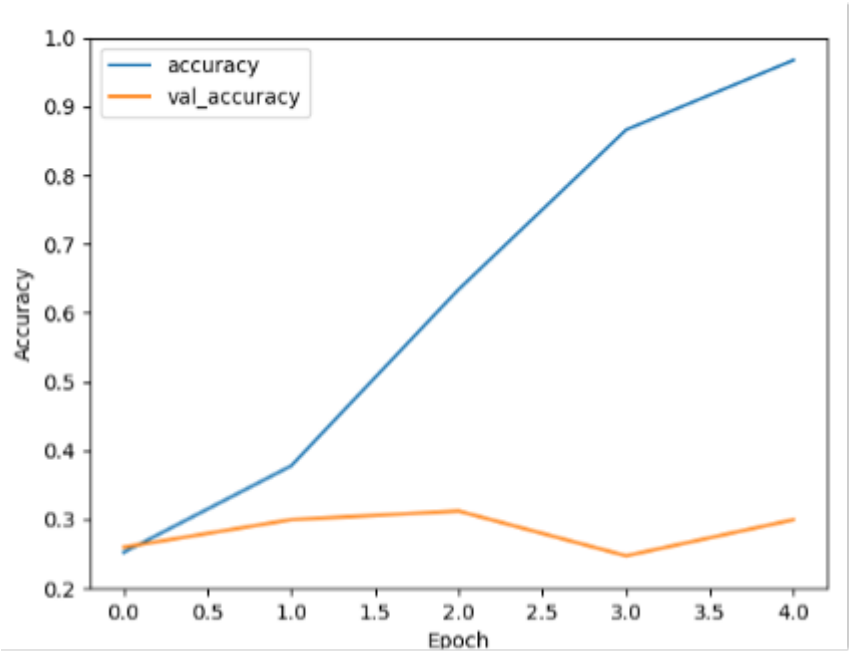

Figure 7: Batch Size =8, taking 38 seconds per epoch.

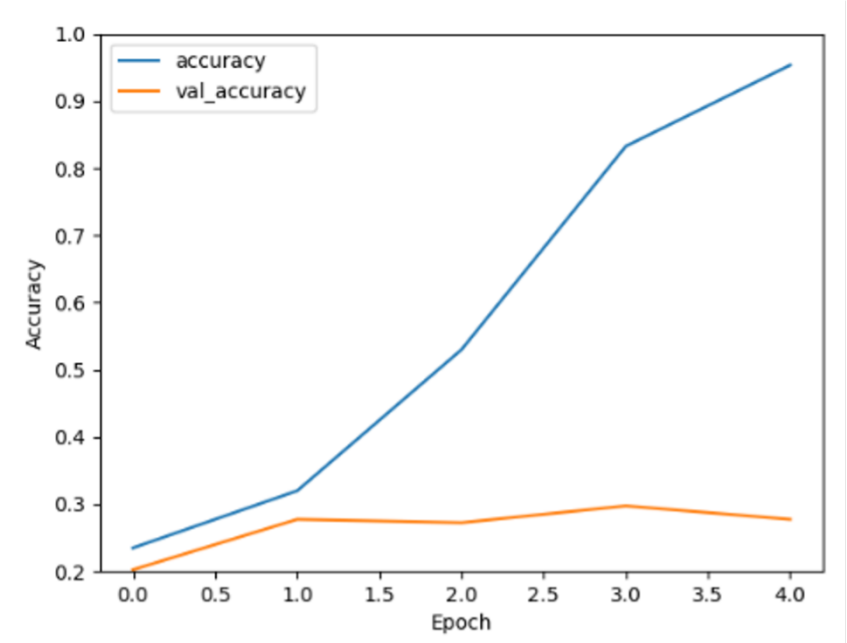

Figure 8: Batch Size =4, taking 49 seconds per epoch.

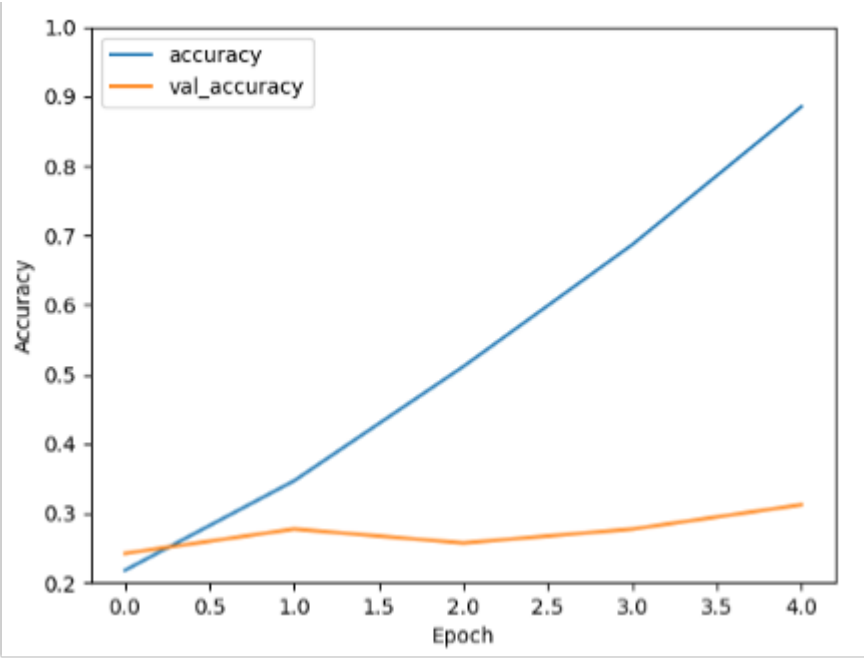

Figure 9: Batch Size =32, taking 31 seconds per epoch.

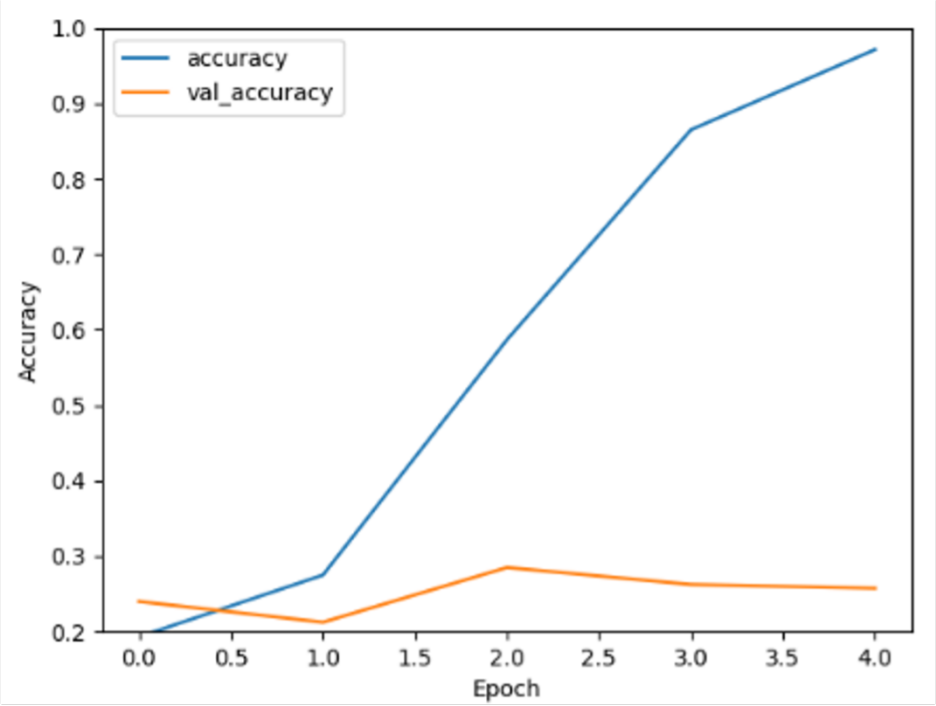

Figure 10: 64,128 Filters. Taking 66 seconds per epoch.

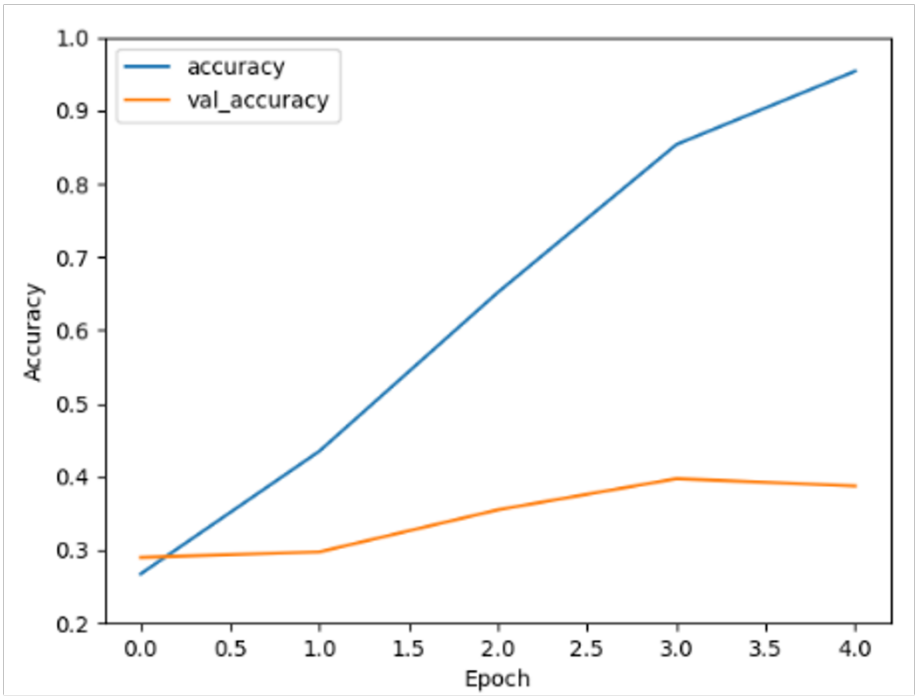

Figure 11: 16,32 Filters. Taking 20 seconds per epoch.

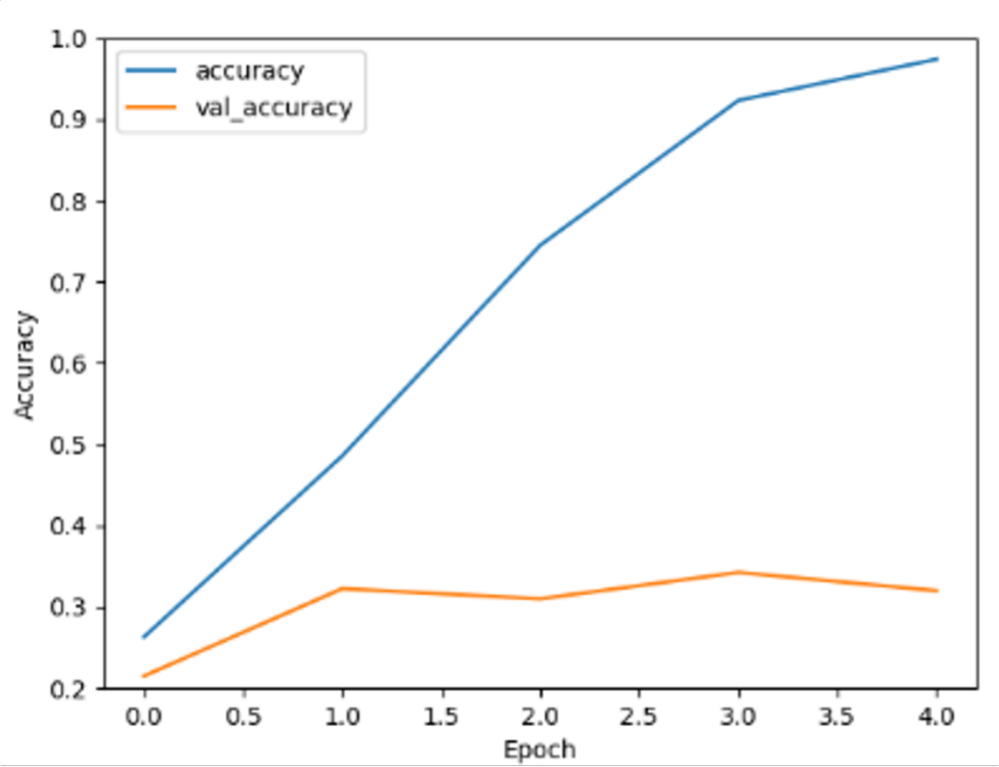

Figure 12: 12,24 Filters. Taking 16 seconds per epoch.

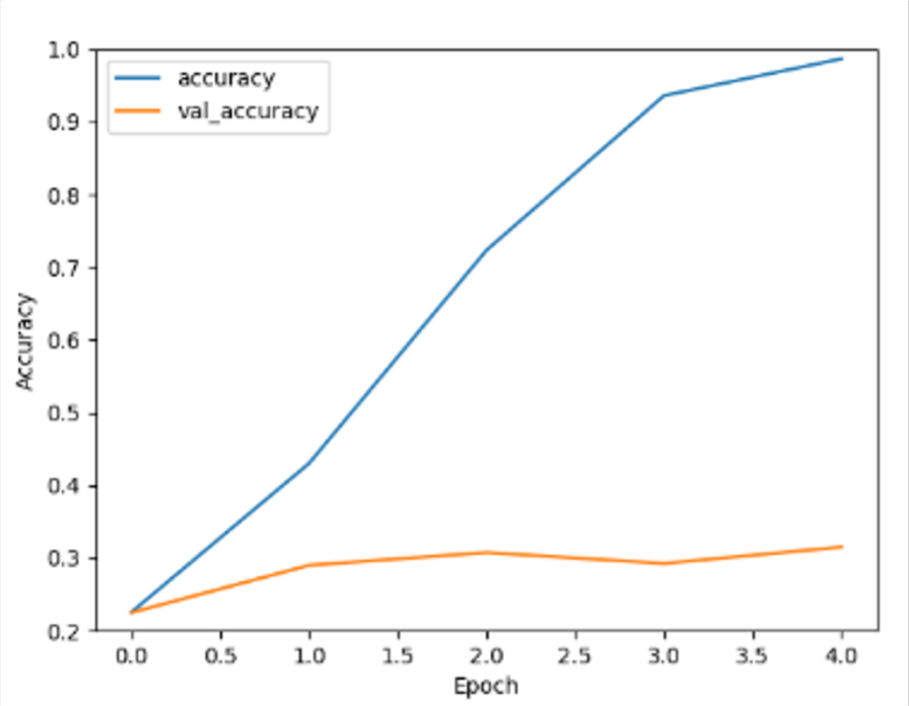

Figure 13: 16,64 Filters. Taking 25 seconds per epoch.

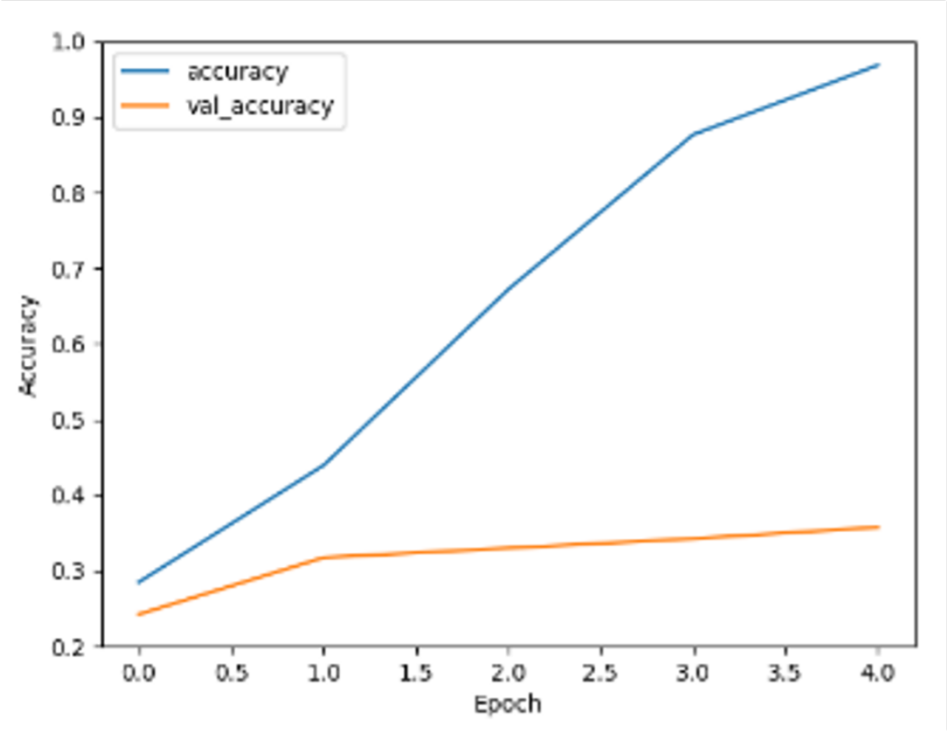

Figure 14: 16,48 Filters. Taking 22 seconds per epoch.

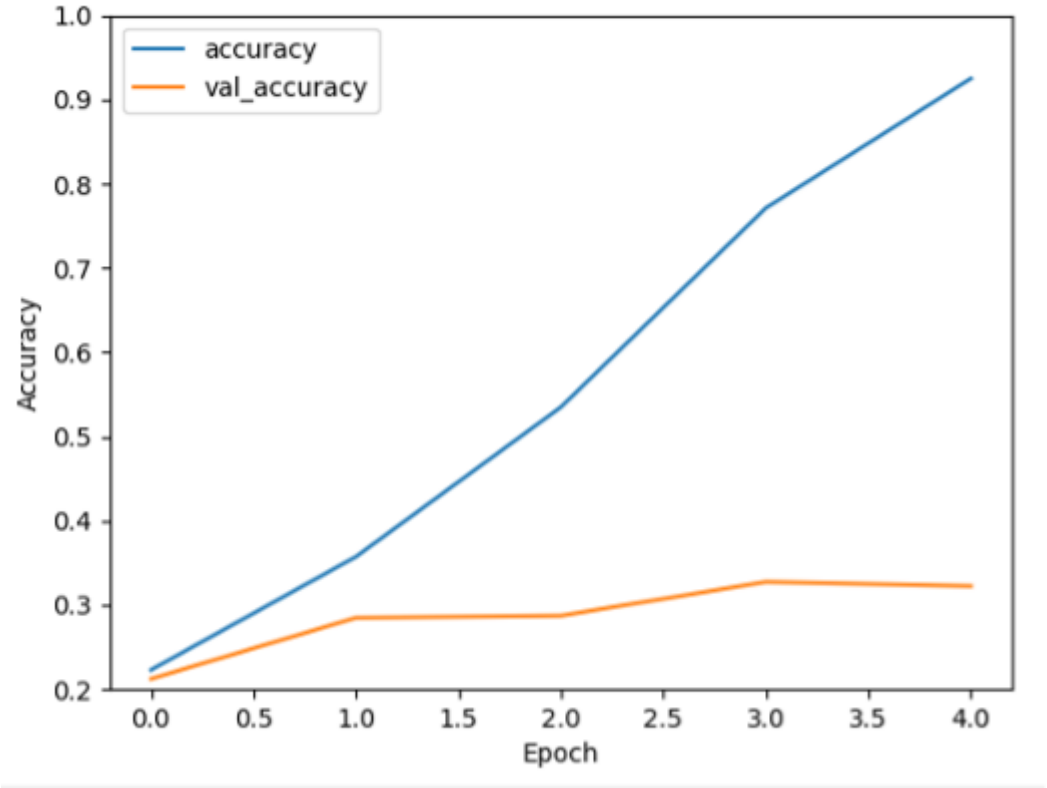

Figure 15: 16,16 Filters. Taking 16 seconds per epoch.

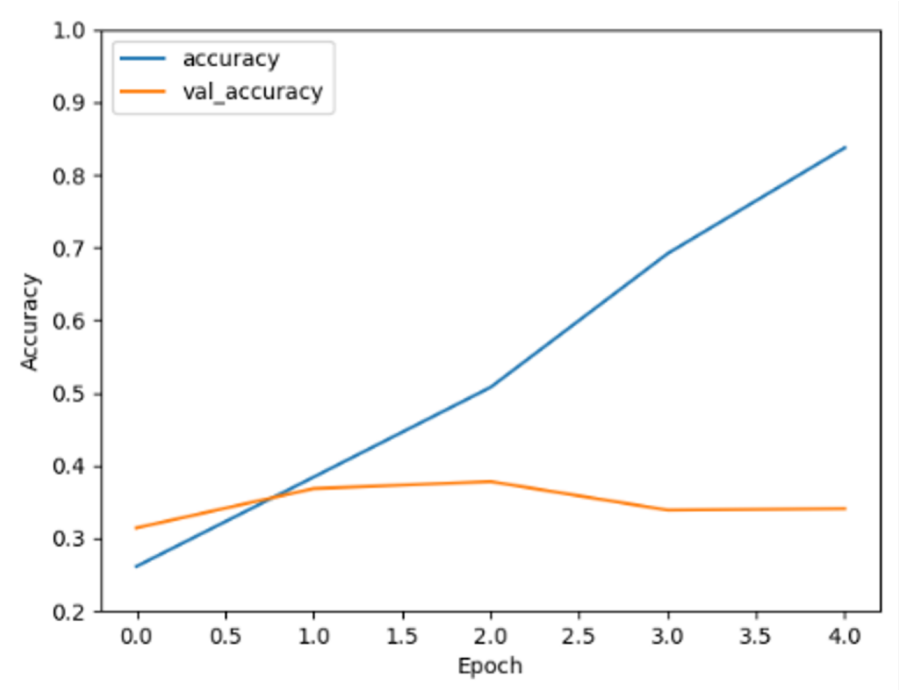

Figure 16: 16,32 Filters. Taking 78 seconds per epoch, on the full dataset.

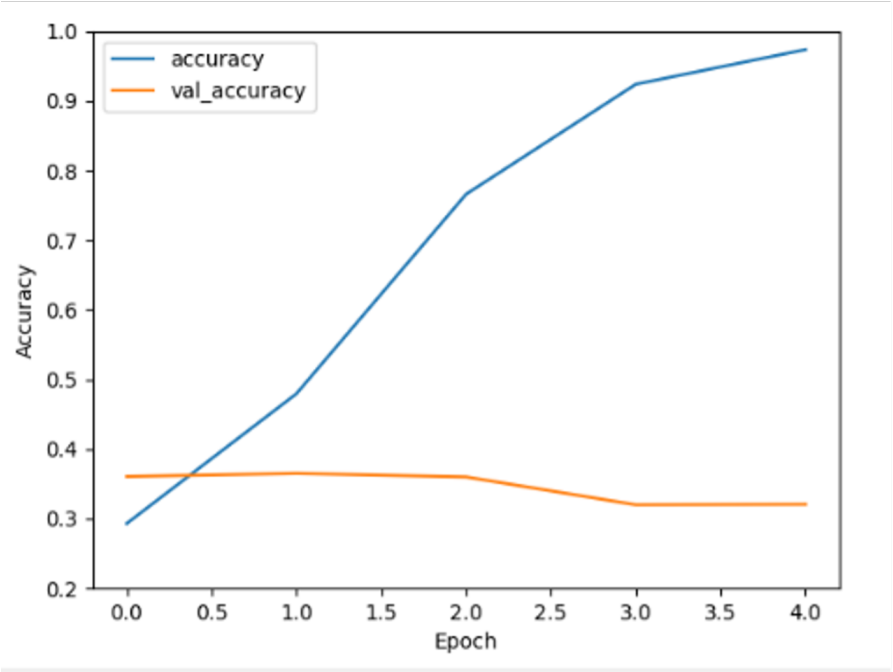

Figure 17: 24,48 Filters. Taking 105 seconds per epoch, on the full dataset.

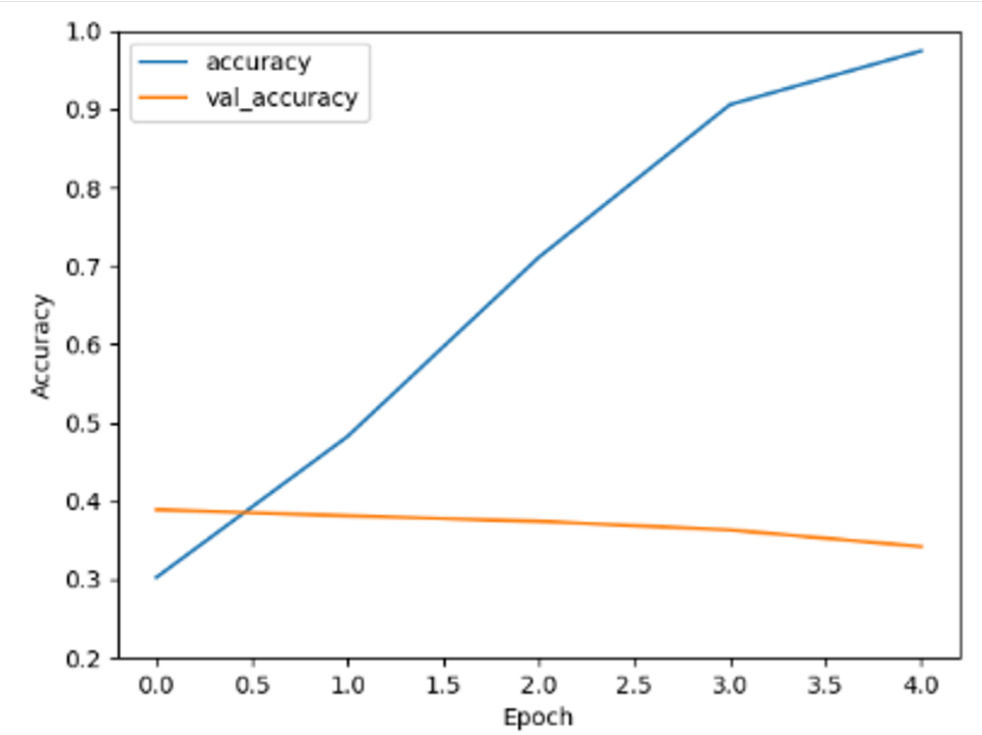

Figure 18: 12,24 Filters. Taking 61 seconds per epoch, on the full dataset.