Muhammad Tanveer

Ken Ko

CSCI 49377

Total Words: 4900

Introduction

We were tasked with building an image classifier that would be able to classify images to 5 classes with above 85% accuracy. The dataset consisted of 22500 images all equally distributed within the following classes: bell, dog, horse, house, and tiger. The dataset was further stratified into train, test, and validate sets.

Our final model achieved 90% accuracy on the holdout test set. In each stage of this project, we iterated upon our best models, making sure that each team member experimented with parallel parameters of the model to ensure that models could be combined at the end of each stage.

Data Preprocessing

For models developed in stages 3 and 4 the ratio of stratification was 80% train set, 10% test set, and 10% validation set. A single instance in the dataset was a 512 x 512 x 3 color image. The images were standardized via Keras ImageDataGenerator such that all the pixels have values between 0 and 1. For any data augmentation the option was selected for missing pixel values to be set to their nearest pixel.

Then the ImageDataGenerator flows were set for each of the training, testing, and validation sets. The parameters for the flow are as follows:

Image_gen.flow_from_directory(set_path, target_size, class_mode, batch_size, shuffle) where:

a. Set_path : the directory path for the set
b. Target_size: the size all images should be resized to; 512 x 512
c. class_mode: the type of label array; in our case we have multiple classes and we will one-hot encode the classes.
d. batch_size: size of batches for the data; default 32 used

e. shuffle: whether to randomly shuffle the data; setting to False sorts the data alphanumerically

As seen in Figure 1, the images were portioned correctly and the categories were identified as expected with the correct number of images belonging to each category.

```
# default batch_size is 32
train_image_gen = image_gen.flow_from_directory(train_path,
                                                target_size=image_shape[:2],
                                                class_mode='categorical',
                                                batch_size=batch_size)


# loads and iterates test dataset
test_image_gen = image_gen.flow_from_directory(test_path,
                                               target_size=image_shape[:2],
                                               class_mode='categorical',
                                               batch_size=batch_size,
                                               shuffle=False)

# loads and iterates validation dataset
val_image_gen = image_gen.flow_from_directory(val_path,
                                              target_size=image_shape[:2],
                                              class_mode='categorical',
                                              batch_size=batch_size,
                                              shuffle=False)
```

```
Found 18000 images belonging to 5 classes.
Found 2250 images belonging to 5 classes.
Found 2250 images belonging to 5 classes.
```

Figure 1 – Preprocessing Dataset with ImageDataGenerator

As a final sanity check each set was inspected for the shape of the images generated to ensure the correct preprocessing was accomplished (see Figure 2). The output tuple indicates the batch size, the image height, the image width, and the image color scheme.

```
for i in range(5):
  X,y = train_image_gen.next()
  print(X.shape)
```

```
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
```

```
for i in range(5):
  X,y = test_image_gen.next()
  print(X.shape)
```

```
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
```

```
for i in range(5):
  X,y = val_image_gen.next()
  print(X.shape)
```

```
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
(32, 512, 512, 3)
```

Figure 2 – Sanity Check for ImageDataGenerator Preprocessing

Stage 3

Baseline (Model M1)

We began with Muhammad's baseline model from Lab Assignment 7, which contained 5 convolutions layers and 3 dense layers. The five convolution layers used filter sizes of [64,32,16,16,16] from earliest layer to latest. A 2x2 max pooling layer was added after every convolution layer. The weights were initialized using the default glorot initialization. ReLU was used as the activation unit for all convolution layers. The following 2 dense layers consisted of 1 layer of 16 units and 1 layer of 8 units. The final dense layer consists of 5 units and a softmax activation function. This architecture is shown in Figure 3.1. The model used the nadam optimization algorithm. The final training accuracy was only 64% with a validation accuracy of 61% and a testing accuracy of 56%. An early stopping parameter (based on minimum validation loss with a patience of 10 epochs) was included in all models but not all models reached an early stopping point. This model tested by Muhammad using a batch size of 32 and by Ken using a batch size of 64. The results of the model using a batch size of 64 is recorded in Table 1, M1 column. However, there was no significant difference in final accuracy or training time b/t the two models. In addition, because a batch size of 64 doubled the model's GPU RAM requirements, all subsequent models used a batch size of 32. The low training accuracy indicated an underfitting issue so we decided to increase the number of layers and units.

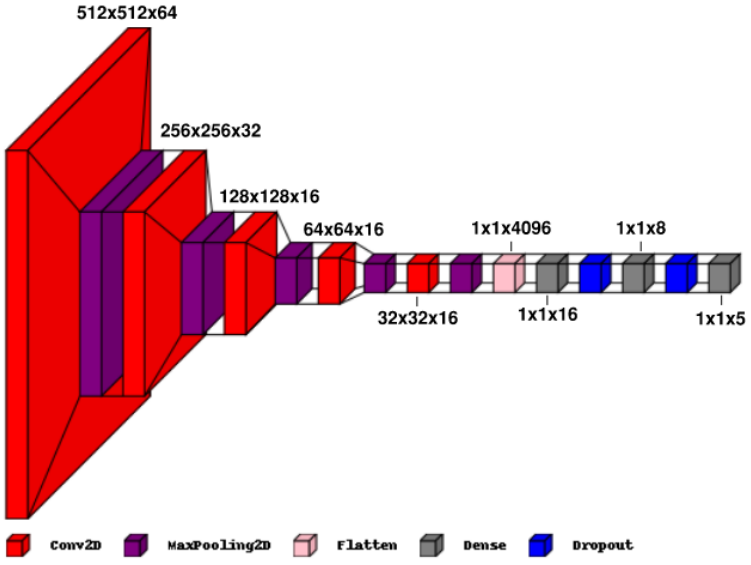

Figure 3.1: M1 Architecture

Model M2

Muhammad experimented with increasing the convolutional layers and filters per layer by increasing the convolutional layers from 5 to 7 and the filter sizes from [64,32,16,16,16] to [128,128,64,64,32,32,16]. Although the training and validation results of the model were unfortunately deleted before they could be recorded, the model produced a final testing accuracy of 72%. The architectural changes made to this model were later brought over to the combined team model. The model architecture can be seen in Figure 3.2.

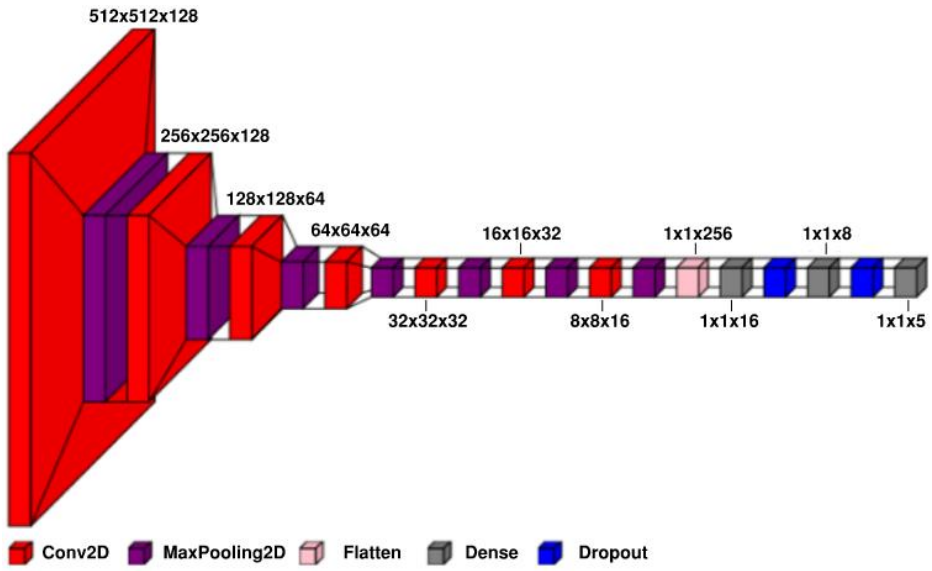

Figure 3.2: M2 Architecture

Model M3

Ken experimented with increasing the number of units in the dense layers from [16,8,5] to [128,64,5] (see Table 1, M3 column). This led to a training accuracy of 73% with a validation accuracy of 73% and a testing accuracy of 72% in the best epoch (epoch 5 out of 15). This model underfits less but increased the amount of overfitting. The model architecture can be seen in Figure 3.3.

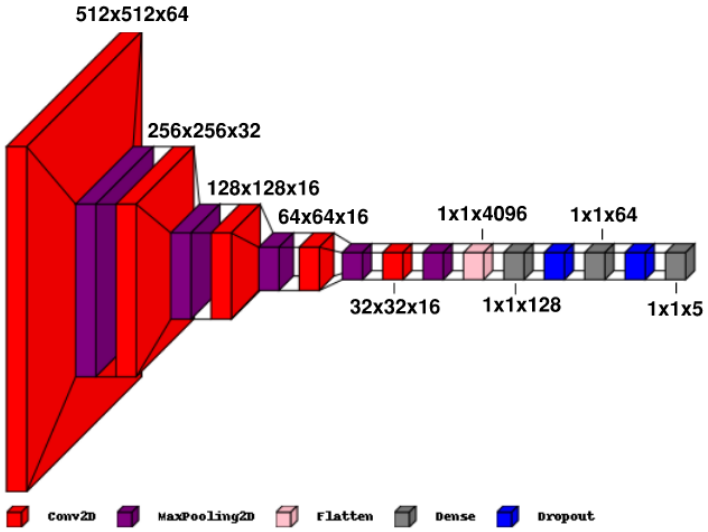

Figure 3.3: M3 Architecture

Model M4

We then tested the combined team model (see model architecture in Figure 3.4). This model combined the changes made in M2 and M3. This led to a training accuracy of 81% with a validation accuracy of 77% and a testing accuracy of 78% in the best epoch (epoch 16 out of 26). Overall, this combined model showed an improvement over all the previous models. The results of this model are shown in Table 1, M4 column.

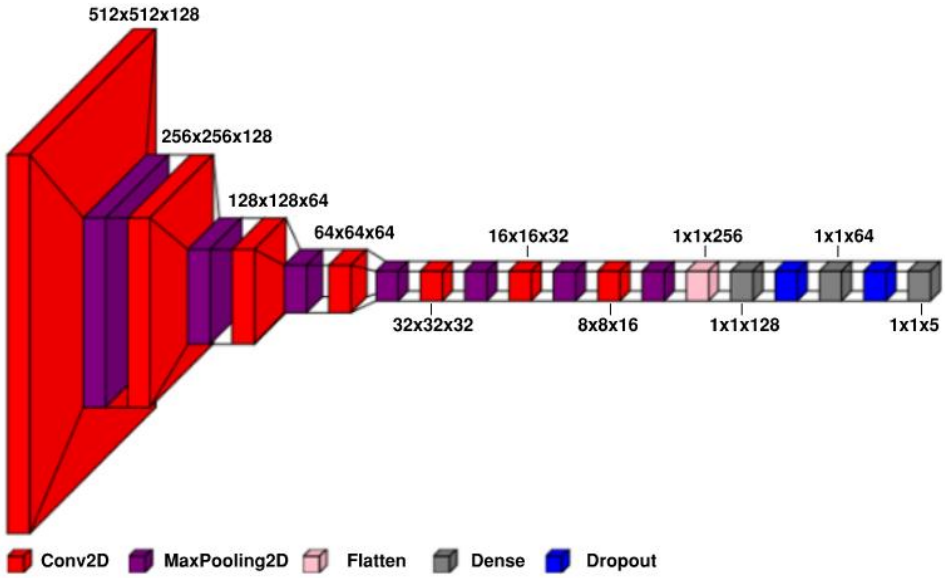

Figure 3.4: M4 and M5 Architecture

Model M5

As a final experiment, we tested the same architecture as model M4 but used the adam optimizer (see model architecture Figure 3.4). It should also be noted that unlike the previous models, model M5 was trained using categorical cross entropy loss function and ImageDataGenerator was modified to produce one-hot encoded labels. This was done to pass Gradescope's autograder. This model resulted in a training accuracy of 77% with a validation accuracy of 75% and a testing accuracy of 75% during the best epoch (epoch 15 out of 25). The results of this model are shown in Table 1, M5 column. Overall, this model performed worse than model M4.

Architecture	M1 (baseline)	M2	M3	M4	M5
Test accuracy	0.5644	0.72	0.7227	0.7769	0.7502
Val accuracy	0.6160	-	0.7284	0.7667	0.7502
Train accuracy	0.6413	-	0.7277	0.8089	0.7742
Optimizer	nadam	nadam	nadam	nadam	**adam**
Batch size	32, 64	32	32	32	32
Activation unit	ReLU	ReLU	ReLU	ReLU	ReLU
Weight init	glorot	glorot	glorot	glorot	glorot
Early stopping	No	No	Yes	Yes	Yes
Epochs	30	30	15	26	25
Dropout rate	0.5	0.5	0.5	0.5	0.5
Padding	same	same	same	same	same
# of filters in convolution layers	[64,32,16, 16,16]	**[128,128,64, 64,32,32,16**	[64,32,16, 16,16]	**[128,128,64, 64,32,32,16]**	[128,128,64, 64,32,32,16]
Kernel size	[5,3,3,3,3]	[5,3,3,3,3,3,3]	[5,3,3,3,3]	[5,3,3,3,3,3,3]	[5,3,3,3,3,3,3]
# of units in dense layers	[16,8,5]	[16,8,5]	**[128,64,5]**	**[128,64,5]**	[128,64,5]
# of trainable params	98,325	304,661	565,589	341,845	341,845
# of non- trainable params	0	0	0	0	0

Table 1: Overview of Stage 3 Models

Based on model testing accuracy in Table 1 above, model M4 performed the best out of the 5 models. From this, we can conclude that increasing the complexity of the baseline model resulted in a sizeable performance increase. However, one major issue with M4 was its tendency to overfit the data. Figure 3.5 below shows a graph of model M4's loss function and accuracy over time during training. Although the loss steadily decreased over time, the validation loss did not decrease much after around epoch 10 and even began to increase in later epochs. Therefore, one of the goals of Stage 4 was to reduce this overfitting.

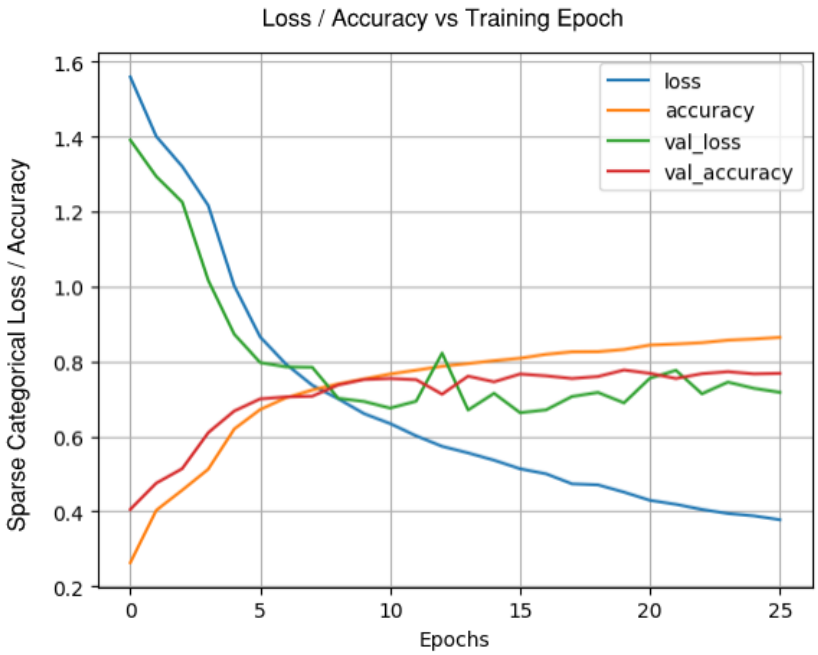

Figure 3.5: M4 Loss / Accuracy Over Time

Stage 4

Experimentation for Stage 4 proceeded in two directions. Models M4 and M5 from Stage 3 showed promise, so Muhammad experimented with using data augmentation and different hyperparameters in order to reduce overfitting. On the other hand, Ken experimented with implementing the skip connections technique used in ResNet, which would allow us to create a very deep model.

Model M5 v3

The previous M5 architecture is shown in Figure 3.4 above. This same architecture was used by Muhammad along with He weight initialization and data augmentation. The parameters and results are recorded in Table 2, column M5 v3 below. This model achieved a training accuracy of 81% and a validation accuracy of 80% during the best epoch (epoch 30 out of 30). It achieved a final test accuracy of 80%. Data augmentation reduced overfitting and allowed the model to train for the full 30 epochs without hitting an early stopping point.

Model M6

Ken worked on improving the architecture by utilizing skip connections. Model M5 was first rebuilt using the Keras functional API. It became apparent that it would be difficult to implement skip connections on this rebuilt model due to each max pooling layer quartering the size of the previous layer. Skipping an output forward in this scenario would lead to shape mismatches between inputs. Thus, model M5 was scrapped and model M6 was created.

Similar to Resnet, model M6 consists of a convolution layer followed by a single 2x2 max pooling layer at the start of the model, followed by 17 convolution layers using the ReLu activation function. The default glorot weight initialization was used. A skip connection was added every 3 convolution layers. To keep the shapes consistent for the skip connections, every convolution layer uses 64 filters and ‘same’ padding. The convolution layers were followed by a global average pooling layer, followed by one dense layer of 256 units using ReLU and a 0.5 dropout rate, followed by one dense layer of 5 units using softmax. The ‘adam’ optimizer was used. The batch size was kept at the default of 32. All layers used ‘same’ padding.

However, this model encountered a bug during training, in which the validation accuracy per epoch remained fixed at 20% and training accuracy did not improve between epochs. The results of this model were not saved, as it was clear that continuing to train the model would not yield any useful results. Applying batch normalization to every third convolutional layer solved this issue. A graphic of the full model M6 architecture is shown in Figure 4.1 below, and the parameters and results are recorded in Table 2, column M6.

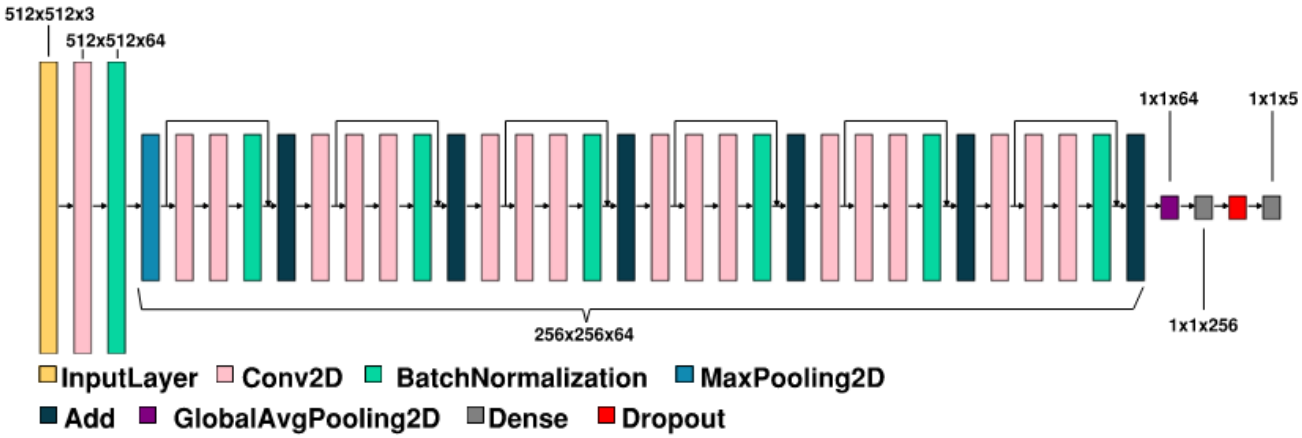

Figure 4.1: Model M6 Architecture

This model achieved a training accuracy of 85% and a validation accuracy of 81% during the best epoch (epoch 19 out of 30). It achieved a final test accuracy of 81%.

Model M7

Model M7 was created by Ken by building upon M6. Previously in M6, batch normalization was applied to every third convolution layer after an activation function was applied. In model M7, batch normalization was applied after every convolution layer and before an activation function was applied. In addition, the number of filters per convolution layer was modified. Instead of every layer having 64 filters, the first 6 layers were given 32 filters, the middle 6 layers were given 64 filters, and the last 6 layers were given 128 filters. This is similar to VGGNet and ResNet which both increase the number of filters on a layer the deeper the level of the layer.

However, this model ran into 2 issues. The first issue was that the increased complexity of the model required more GPU RAM than Colab could provide, even with the highest amount available to Colab Pro. To fix this, the batch size was decreased from 32 to 16. This fixed the RAM issue but made it harder for the model to converge onto a solution. The second issue was

that after 3 training epochs, validation accuracy dropped back down to 20% and training halted. It's unknown what exactly causes this issue to keep popping up, but one possible explanation is that some combination of vanishing gradients and negative weights passed into the ReLU activation function was causing the model to always output 0. To fix this, the optimizer was changed from 'adam' to 'nadam' in hopes that the added momentum term would prevent this behavior. The final model M7 architecture is shown in Figure 4.2 below. The parameters and results are recorded in Table 2, column M7 in the next section. This model contained 1,100,005 trainable parameters and 2,112 non-trainable parameters.

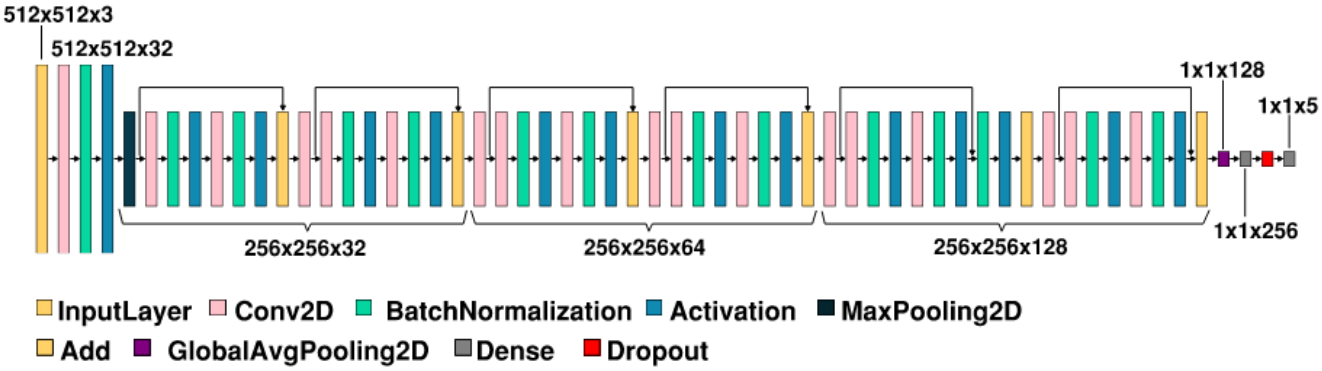

Figure 4.2: Model M7 Architecture

This model achieved a training accuracy of 77% and a validation accuracy of 72% during the best epoch (epoch 14 out of 30). It achieved a final test accuracy of 73%. However, the validation accuracy bounced around quite a bit due to the smaller batch size (see Figure 4.3). In addition, from epoch 16 on, the model once again ran into the issue in which validation accuracy dropped to 0.2000 and stayed there for every subsequent epoch. This suggests that increasing the model complexity further may not be the way to go.

Figure 4.3: M7 Loss / Accuracy Over Time

Model M8

Model M8 was created by Ken by comparing the results from M6 and M7. The comparison indicated that a batch size of 32 was more ideal than 16, along with a less complex model. Therefore, the number of filters per convolution layer was changed back to 64, as in model M6. This brought down the GPU RAM requirements such that a batch size of 32 could be used again. Batch normalization was placed after each convolution layer, as in model M7. Additional batch normalization layers were also added after each skip connection (see Figure 6). In addition, the activation function was changed from 'ReLU' to 'Leaky ReLU'. This was done in case the past issue in which the model is stuck at 20% validation accuracy is caused by some sort of dying ReLU problem. The alpha term in the Leaky ReLU layer was kept at the default of 0.3. A typical portion of the model is shown in Figure 4.4.

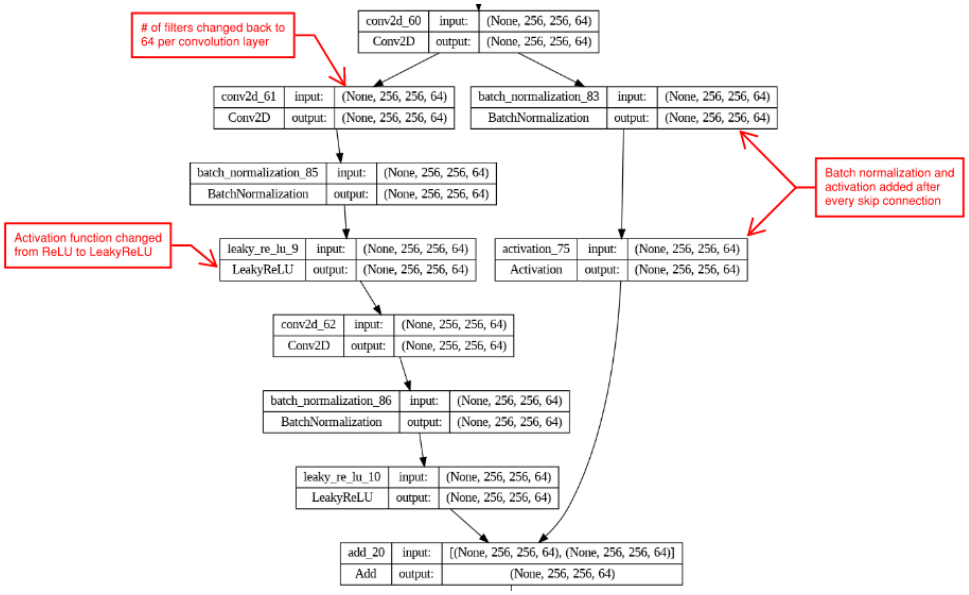

Figure 4.4: Typical Portion of Model M8

The final model M8 architecture is shown in Figure 4.5 below. The parameters and results are recorded in Table 2, column M8 in the next section. This model contained 653,125 trainable parameters and 2,560 non-trainable parameters.

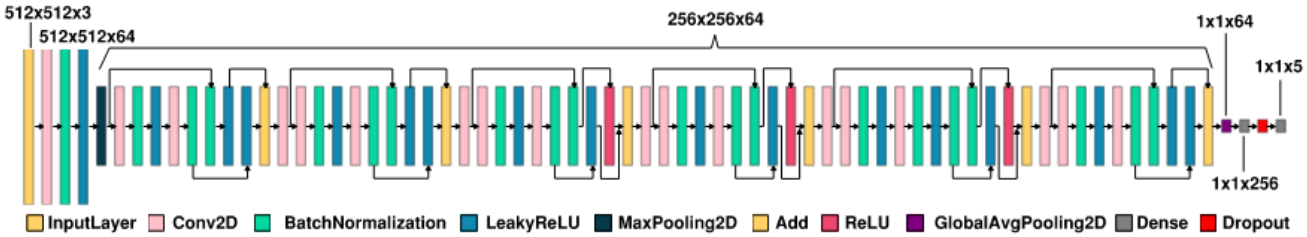

Figure 4.5: Model M8 Architecture

This model achieved a training accuracy of 83% and a validation accuracy of 76% during the best epoch (epoch 30 out of 30). It achieved a final test accuracy of 76%.

Model M9

Muhammad created Model M9 by combining the architecture of M8 with the weight initialization and data augmentation of M5 v3. It was noticed that model M8 had some mistakes in the architecture. The first was that the model contained a batch normalization + activation layer followed by another batch normalization + activation layer.

The extra batch normalization + activation layer was removed in model M9. The second mistake was that in switching from ReLU to LeakyReLU, some units were missed and remained as ReLU. This was also changed in model M9 so that all layers used LeakyReLU. In addition, the weights in every convolution layer were previously initialized using Glorot initialization and changed to now be initialized using He initialization. ImageDataGenerator was also modified to provide data augmentation.

The final model M9 architecture is shown in Figure 4.6 below. The parameters and results are recorded in Table 2, column M9 in the next section. This model contained 652,997 trainable parameters and 2,432 non-trainable parameters.

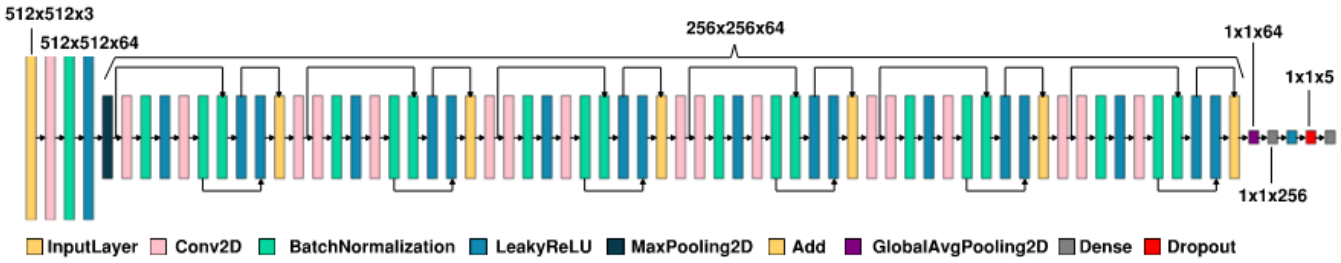

Figure 4.6: Model M9 Architecture

This model achieved a training accuracy of 74% and a validation accuracy of 74% during the best epoch (epoch 27 out of 30). It achieved a final test accuracy of 64%. Interestingly, even though data augmentation was used, this model still overfit and did not perform as well as M6.

Architecture	M5 v3	M6	M7	M8	M9
Test accuracy	0.8004	0.8089	0.7289	0.7596	0.6427
Val accuracy	0.7951	0.8133	0.7231	0.7582	0.6360
Train accuracy	0.8108	0.8501	0.7696	0.8326	0.7467
Optimizer	adam	adam	**nadam**	nadam	nadam
Batch size	32	32	**16**	**32**	32
Activation unit	ReLU	ReLU	ReLU	**LeakyReLU**	LeakyReLU
Weight init	He	**glorot**	glorot	glorot	**He**
Early stopping	no	yes	yes	no	no
Epochs	30	29	24	30	30
Dropout rate	0.5	0.5	0.5	0.5	0.5
Skip connections	no	**yes**	yes	yes	yes
Batch normalization	yes	yes	yes	yes	yes
Data augment	yes	no	no	no	**yes**
# of conv layers	18	18	18	18	18
# of filters in conv layers	[128,128,64, 64,32,32,16]	64 for all 18 layers	**32 for first 6, 64 for middle 6, 128 for last 6**	**64 for all 18 layers**	64 for all 18 layers
Kernel size	5 for first conv, 3 for all others	5 for first conv, 3 for all others	5 for first conv, 3 for all others	5 for first conv, 3 for all others	5 for first conv, 3 for all others
# of units in dense layers	[128,64,5]	[256,5]	[256,5]	[256,5]	[256,5]
# of trainable params	342,773	651,461	1,100,005	653,125	652,997
# of non-trainable params	928	896	2,112	2,560	2,432

Table 2: Overview of Stage 4 Models

As seen in Table 2, model M6 with the new skip connections outperformed the other models, even without data augmentation. Thus, Stage 5 will focus on further refining this architecture.

Stage 5

Model M10

Using suggestions from Professor Ligorio, model M10 was created by Ken. The model consists of a convolution layer followed by a single 2x2 max pooling layer at the start of the model, followed by 17 convolution layers using the ReLu activation function. Each convolution layer used a filter size of 64 and He weight initialization. A skip connection was added every 3 convolution layers. Each skip connection is followed by a batch normalization layer and a 2x2 max pooling layer. The first convolution layer uses a 5x5 kernel, and the other layers use a 3x3 kernel. The convolution layers were then followed by a global average pooling layer, followed by a single dense layer of 5 units using softmax. The 'adam' optimizer was used. The batch size was kept at the default of 32. All layers used 'same' padding.

A graphic of the full model M10 architecture is shown in Figure 5.1 below, and the parameters and results are recorded in Table 3a, column M10. This model contained 633,861 trainable parameters and 896 non-trainable parameters.

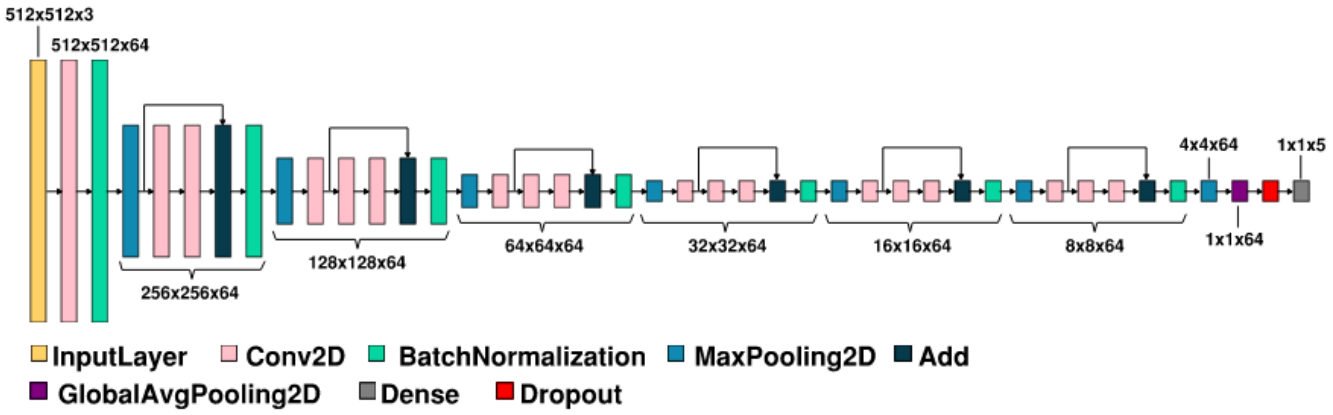

Figure 5.1: Model M10 Architecture

This model achieved a training accuracy of 89% and a validation accuracy of 84% during the best epoch (epoch 10 out of 30). It achieved a final test accuracy of 86%.

Model M11

Model M11 was created by Ken by making a slight change to model M10. In the previous model, each convolution layer had 64 filters per layer. In this model, the first six layers have 32 filters each, the middle six layers have 64 filters each, and the last 6 layers have 128 filters each. A

graphic of the full model M11 architecture is shown in Figure 5.2 below, and the parameters and results are recorded in Table 3a, column M11 in the next section. This model contained 1,065,189 trainable parameters and 960 non-trainable parameters.

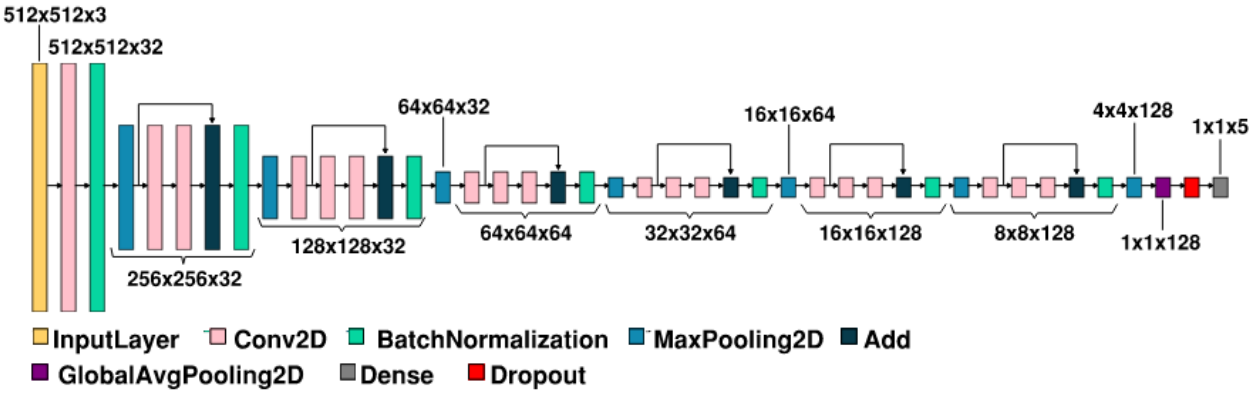

Figure 5.2: Models M11-M15 Architecture

This model achieved a training accuracy of 89% and a validation accuracy of 84% during the best epoch (epoch 9 out of 19). It achieved a final test accuracy of 86%. Overall, it offered a very small improvement over model M10.

Model M12

Model M12 was created by Ken and uses the same architecture as model M11. However, each convolution layer's activation function was changed from 'relu' to 'swish'. See Figure 5.2 for the model's architecture. This model contained 1,065,189 trainable parameters and 960 non-trainable parameters. Interestingly, this change reduced the model's GPU RAM usage from 12 gb to 8 gb, allowing us to save on compute units. The parameters and results are recorded in Table 3a, column M12 in the next section. This model contained 1,065,189 trainable parameters and 960 non-trainable parameters. This model achieved a training accuracy of 95% and a validation accuracy of 85% during the best epoch (epoch 15 out of 25). It achieved a final test accuracy of 85%.

Model M13

Model M12 offered the same performance as M11 while using fewer resource. Thus, Model M13 was created by Ken using the same architecture and activation function as model M12 with the addition of data augmentation. The ImageDataGenerator was modified to provide data augmentation. See Figure 5.2 for the model's architecture. It was first trained for 30 epochs using

an early stopping callback with a patience of 10 epochs. Because it did not reach an early stopping point after 30 epochs and accuracy was slowly improving, the model was trained for an additional 30 epochs. This time, the model hit an early stopping point after 25 epochs. The parameters and results are recorded in Table 3a, column M13 in the next section. This model contained 1,065,189 trainable parameters and 960 non-trainable parameters.

This model achieved a training accuracy of 91% and a validation accuracy of 89% during the best epoch (epoch 45 out of 55). It achieved a final test accuracy of 88%. The benefits of data augmentation here are clear, as the amount of overfitting was greatly reduced, and the final test accuracy was improved. On the other hand, using data augmentation greatly increased the training time (this model took almost 30 hrs to finish training!). The model also needed to be trained over 2 separate runtimes. Therefore, data augmentation will only be used again for the final project model and not any other test models.

Model M14

This model was created by Ken in an attempt to reduce overfitting without using data augmentation to avoid long training times. It uses the same architecture as M13 (see Figure 5.2) but adds an L2 regularization value of 0.0005 to every convolution layer. The parameters and results are recorded in Table 3a, column M14 in the next section. This model contained 1,065,189 trainable parameters and 960 non-trainable parameters.

This model achieved a training accuracy of 81% and a validation accuracy of 79% during the best epoch (epoch 6 out of 16). It achieved a final test accuracy of 79%. This did reduce the overfitting but negatively impacted the overall accuracy of the model.

Model M15

This model attempts the same thing as M14 but uses a larger L2 regularization value of 0.005 instead. It uses the same architecture as M13 (see Figure 5.2) but adds an L2 regularization value of 0.0005 to every convolution layer. The parameters and results are recorded in Table 3b, column M15 in the next section. This model contained 1,065,189 trainable parameters and 960 non-trainable parameters.

This model achieved a training accuracy of 78% and a validation accuracy of 74% during the best epoch (epoch 7 out of 17). It achieved a final test accuracy of 75%. Increasing the L2 regularization to 0.005 caused the model to perform even worse than before. Therefore, this hyperparameter will be ignored and the final model will not contain any regularization.

Model M16

This model was created by Muhammad to experiment with different architectures. In previous models, the first skip connection combines the output of a MaxPooling2D layer with the output of a Conv2D layer. All subsequent skip connections combined the output of a Conv2D layer with the output of a Conv2D layer (see Figure 5.2). Model M16 changes this architecture of the previous models slightly so that all skip connections combine the output of a MaxPooling2D layer with the output of a Conv2D layer (see Figure 5.3). Because the skip connections no longer start at the Conv2D layers, using a different number of filters per Conv2D layer would lead to shape mismatches so for convenience, all layers were changed back to using 64 filters. A graphic of the full model M16 architecture is shown in Figure 5.6 below, and the parameters and results are recorded in Table 3b, column M16 in the next section. This model contained 633,861 trainable parameters and 896 non-trainable parameters.

This model achieved a training accuracy of 91% and a validation accuracy of 85% during the best epoch (epoch 11 out of 21). It achieved a final test accuracy of 86%. Surprisingly, this model performed better than the other non-data augmented models.

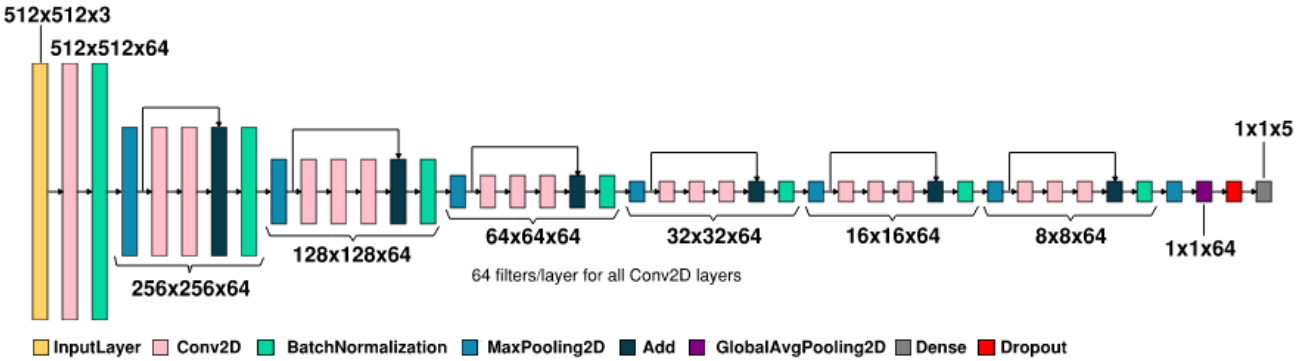

Figure 5.3: Model M16 Architecture

Model M17

Model M17 was created by Muhammad to simultaneously explores the effect of using depthwise separable convolution layers along with deeper layers. The hope is that using depthwise separable layers will reduce the training time while maintaining close to the same amount of performance. Using deeper layers is meant to possibly improve the model's performance and reduce underfitting. The model is based on the architecture of model M12. It consists of a depthwise separable convolution layer followed by a single 2x2 max pooling layer at the start of the model, followed by 20 depth-wise separable convolution layers using the swish activation function. The first six layers have 32 filters each, the middle six layers have 64 filters each, and the last 9 layers have 128 filters each. A skip connection was added every 3 convolution layers. The first skip connection combines the output of a MaxPooling2D layer with the output of a SeparableConv2D layer. All subsequent skip connections combine the output of a SeparableConv2D layer with the output of a SeparableConv2D layer. Each skip connection is followed by a batch normalization layer and a 2x2 max pooling layer. The first convolution layer uses a 5x5 kernel, and the other layers use a 3x3 kernel.

A graphic of the full model M17 architecture is shown in Figure 5.4 below, and the parameters and results are recorded in Table 3b, column M17 in the next section. This model contained 185,072 trainable parameters and 1,216 non-trainable parameters.

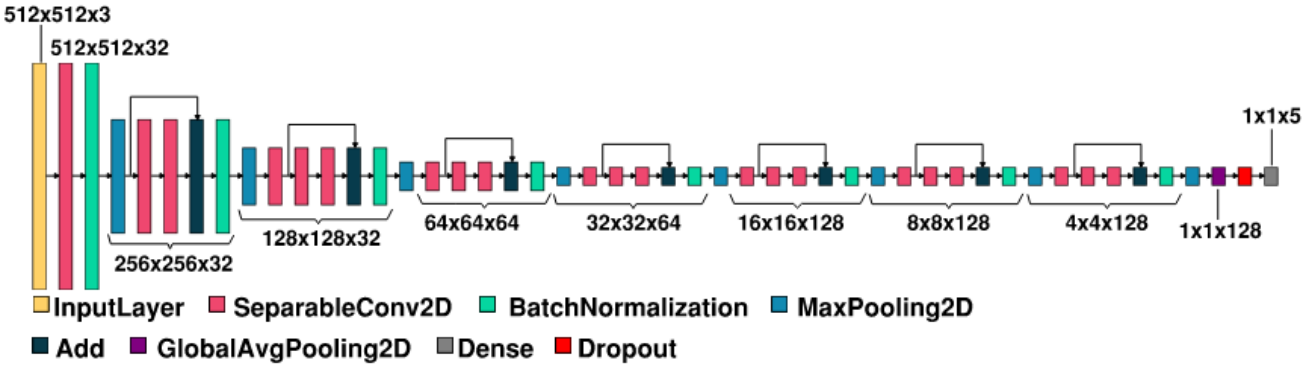

Figure 5.4: Model M17 Architecture

This model achieved a training accuracy of 87% and a validation accuracy of 85% during the best epoch (epoch 9 out of 11). It achieved a final test accuracy of 81%. Each epoch took an average of 511.8 seconds to train.

Model M18

In order to test the effect of using depthwise separable convolutions, every SeparableConv2D layer in model M17 was changed by Muhammad to a Conv2D layer. See Figure 5.5 for the architecture of model M18. The parameters and results are recorded in Table 3b, column M18 in the next section. This model contained 1,508,197 trainable parameters and 1,216 non-trainable parameters.

This model achieved a training accuracy of 93% and a validation accuracy of 80% during the best epoch (epoch 11 out of 21). It achieved a final test accuracy of 83%. Each epoch took an average of 523 seconds to train. When using the depthwise separable convolutions of M17, training time decreased by ~2%. However, performance also decreased by ~2%. Given that the goal of the project is to increase performance as much as possible, this is not enough of a time saving to justify using depthwise convolutions in the final model. Increasing the model depth from 18 to 21 layers also negatively impacted performance (compared to M12).

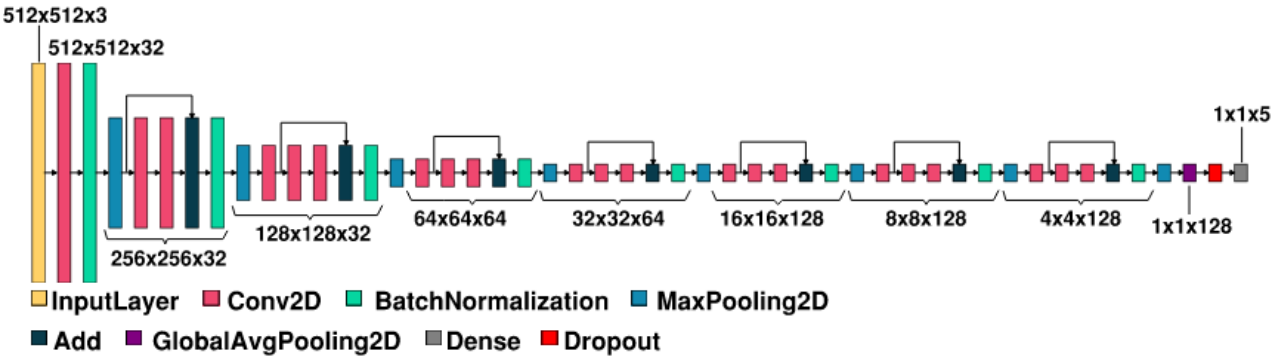

Figure 5.5: Model M18 Architecture

Model M19

Model M19 was one of the two models tested on the entire dataset. It used a 90/10 training/validation set split. The reasoning for this split was that the previous models were trained on 80% of the dataset. Given new data, we would prefer to use the extra data to increase the training set and reduce overfitting.

The architecture used was the same as in model M16 (see Figures 5.3). M16 performed the best out of the Stage 5 non-data augmented models. The hope is that by running the same model with data augmentation, it will perform even better. Since data augmentation proved to be effective in

reducing overfitting in model M13, this model was run without an early stopping point. Instead, a model checkpoint was used to save the model with the lowest validation loss. Due to the long training time, training was split into two different runtimes. During the first training sessions, the model was trained for 60 epochs. This was immediately followed by another 100 epochs of training. Unfortunately, the model was only able to train 34 out of 100 epochs in the second training session before the runtime disconnected. Since the validation loss hadn't improved for 16 epochs by this point and the validation accuracy was also staying constant, training was stopped at this point.

One significant change in this model was made by moving the data augmentation preprocessing step to the model instead of using ImageDataGenerator (see Figure 5.6). Using ImageDataGenerator for data augmentation took a long time because the notebook was using the CPU to perform the image transformations. Instead, various data augmentation layers were added to the model itself to transform the input. This allowed the image transformations to be performed by the GPU, which allowed the model to train in as little as one-third of the time as before.

The parameters and results are recorded in Table 3c, column M19 in the next section. This model contained 633,861 trainable parameters and 896 non-trainable parameters. This model achieved a training accuracy of 91% and a validation accuracy of 89% during the best epoch (epoch 78 out of 94).

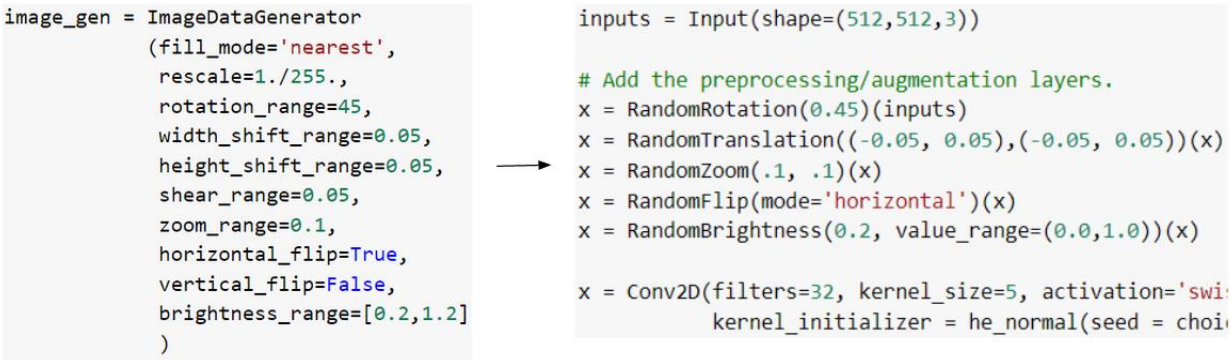

Figure 5.6: Replacement of ImageDataGenerator data augmentation with Data Augmentation layers using Keras

Model M20

Model M20 was also tested on the entire dataset. Like Model M19, it also used a 90/10 training/validation split and data augmentation layers in the model. The architecture used was the same as in model M13 (see Figure 5.2). The reasoning is that model M13 performed fairly well with data augmentation so it should perform even better using the full dataset. One change made in this model was the use of two model checkpoints. One checkpoint was used to save the model with the lowest validation loss and another was used to save the model with the highest accuracy. This is because it is hard to predict which of the two will lead to a higher test accuracy. Therefore, both were saved and tested.

The parameters and results are recorded in Table 3c, column M20 in the next section. This model contained 1,065,189 trainable parameters and 960 non-trainable parameters. The model was first trained for 50 epochs. Then, because it did not hit an early stopping point, it was trained for another 36 epochs before it hit an early stopping point. A patience of 15 epochs was used for early stopping. The reasoning was that using data augmentation would curb overfitting so the early stopping criteria could be loosened.

This model achieved a training accuracy of 90% and a validation accuracy of 88% during the epoch with the lowest validation loss (epoch 71 out of 86). This model achieved a training accuracy of 92% and a validation accuracy of 89% during the epoch with the highest validation accuracy (epoch 85 out of 86). Of the two checkpoints, the latter resulted in a higher accuracy on Gradescope's test set. In addition, although the data augmentation layers in the model are by default only applied during training and not during inference, just in case, they were turned off in the submitted model.

Figures 5.7 and 5.8 below shows a graph of model M20's loss function and accuracy over time during training. As seen, loss continued to decrease over time, even after 86 epochs. On the other hand, although data augmentation was very effective in curbing overfitting, the validation loss seemed to hold more or less steady in the final epochs, suggesting that there is a limit to how much improvement we can get simply by increasing number of training epochs.

Figure 5.7: M20 Loss / Accuracy Over Epochs 1-50

Figure 5.8: M20 Loss / Accuracy Over Epochs 51-86

Architecture	M10	M11	M12	M13	M14
Test accuracy	0.8564	0.8569	0.8533	0.8787	0.7938
Val accuracy	0.8440	0.8449	0.8484	0.8911	0.7911
Train accuracy	0.8899	0.8906	0.9511	0.9124	0.8066
Optimizer	adam	adam	adam	adam	adam
Batch size	32	32	32	32	32
Activation unit	relu	relu	**swish**	swish	swish
Weight init	He	He	He	He	He
Early stopping	yes	yes	yes	yes	yes
Epochs	20	19	25	55	16
Dropout rate	0.5	0.5	0.5	0.5	0.5
L2 regularization	0	0	0	0	**0.0005**
Data augment	no	no	no	**yes**	**no**
# of conv layers	18	18	18	18	18
# of filters in conv layers	64 for all 18 layers	**32 for first 6, 64 for middle 6, 128 for last 6**	32 for first 6, 64 for middle 6, 128 for last 6	32 for first 6, 64 for middle 6, 128 for last 6	32 for first 6, 64 for middle 6, 128 for last 6
Kernel size	5 for first conv, 3 for all others	5 for first conv, 3 for all others	5 for first conv, 3 for all others	5 for first conv, 3 for all others	5 for first conv, 3 for all others
# of units in dense layers	[5]	[5]	[5]	[5]	[5]
# of trainable params	633,861	1,065,189	1,065,189	1,065,189	1,065,189
# of non-trainable params	896	960	960	960	960

Table 3a: Overview of Stage 5 Models

Architecture	M15	M16	M17	M18
Test accuracy	0.7516	0.8636	0.8111	0.8320
Val accuracy	0.7427	0.8516	0.8040	0.8480
Train accuracy	0.7763	0.9124	0.8706	0.9276
Optimizer	adam	adam	adam	adam
Batch size	32	32	32	32
Activation unit	swish	swish	swish	swish
Weight init	He	He	He	He
Early stopping	yes	yes	yes	yes
Epochs	17	21	19	21
Dropout rate	0.5	0.5	0.5	0.5
L2 regularization	**0.005**	**0**	**0**	**0**
Data augment	no	no	no	no
# of conv layers	18	18	**21** (depthwise separable conv)	**21** (regular conv)
# of filters in conv layers	32 for first 6, 64 for middle 6, 128 for last 6	**64 for all layers**	**32 for first 6, 64 for middle 6, 128 for last 9**	32 for first 6, 64 for middle 6, 128 for last 12
Kernel size	5 for first conv, 3 for all others	5 for first conv, 3 for all others	5 for first conv, 3 for all others	5 for first conv, 3 for all others
# of units in dense layers	[5]	[5]	[5]	[5]
# of trainable params	1,065,189	633,861	185,072	1,508,197
# of non-trainable params	960	896	1,216	1,216

Table 3b: Overview of Stage 5 Models

Architecture	M19	M20
Val accuracy	0.8876	0.8844
Train accuracy	0.9088	0.9035
Train/val split	90/10	90/10
Optimizer	adam	adam
Batch size	32	32
Activation unit	swish	swish
Weight init	He	He
Early stopping	no	yes
Epochs	94	86
Dropout rate	0.5	0.5
L2 regularization	**0**	0
Data augment	**yes**	yes
# of conv layers	18	18
# of filters in conv layers	64 for all layers	32 for first 6, 64 for middle 6, 128 for last 6
Kernel size	5 for first conv, 3 for all others	5 for first conv, 3 for all others
# of units in dense layers	[5]	[5]
# of trainable params	633,861	1,065,189
# of non-trainable params	896	960

Table 3c: Overview of Stage 5 Models Trained on Full Dataset