COSC420 Assignment 1

For this assignment, I will be training neural networks using the Oxford Flowers Dataset. In this assignment, there is a course dataset which contains a subset of 10 flowers and a fine dataset which has all 102 flowers. Results will be shown for both datasets.

Task 1

The first task required us to train a Neural Network from scratch (No transfer learning) on the dataset. Conditions are that we keep it under 15 million parameters.

When creating my CNN, I took heavy inspiration from the VGG16 Architectural style. With 5 blocks of Conv2d Layers with blocks 1,2,4 having 2 conv layers and 3,5 having 3 conv layers with the filters starting at 64 and doubling for each layer and finally 2 dense layers at the end. This is essentially a VGG16 model adapted to under 15 million parameters. I would prefer having 3 Conv2d layers for block 4 but that surpassed the parameter limit. I did this as this style of Architecture has been proven to be successful and would serve as a very good baseline for my model. I also had a callback to store the best performer on validation data and load the model in right at the end. As we are testing the best performance on test data and through the epochs, the highest epoch doesn't always mean the best performing model. So loading the highest performance model will give us the optimal model on generalisation.

An issue that I started off with was using the base adam optimizer, my model would get stuck on 6.25% validation accuracy. This was likely because the learning rate for base adam was too high, and it failed to decrease the loss function therefore not changing the validation accuracy (getting stuck). So, to combat this, I decreased the learning rate to 1e-4. This allowed the neural network to properly learn and drop the loss function. As it would stop bouncing around the trough of the derivative of the loss function and actually enter it.

I started with the base VGG Architectural style and added some various things to see how it affects the performance of the models. I added things such as Data Augmentation, Batch Normalization after each conv layer and 0.2 Dropout After each Maxpooling Layer. I chose this amount of data augmentation and to put the layers where they were because these were where they were where they were used commonly from other material online. Below is a table with various test and layers added to see how they would affect the performance of the model, and which are the best performers. I did the test below by training the model to 100 epochs and testing the accuracy of the best performing models.

Model	Test Accuracy
Model no add Ons	57.8%
Model with Data Aug	75.3%
Model with Data Aug + Batch Norm	84.2%
Model with Data Aug + Batch Norm + Dropout	75.9%
Model with Data Aug + Dropout	78.1%

As you can see from the table above the VGG16 Architecture including data augmentation and batch normalization performs the best. So that is what the architecture I will be using in the following models. To try see how far we can take this model I have decided to train the model for a total of 400 epochs. Another interesting thing to look at would be the class imbalance in the data set. In the course train dataset, the imbalance is shown below.

Class 0 = 6.09%

Class 1 = 1.06%

Class 2 = 8.68%

Class 3 = 15.51%

Class 4 = 19.50%

Class 5 = 10.64%

Class 6 = 7.21%

Class 7 = 5.45%

Class 8 = 15.41%

Class 9 = 5.14%

To try combat this, I trained up another model with everything the same, except it accounts for class weights using the fit functions class_weights parameter. Below are the accuracy scores with the accuracy being not accounting for class balance and the overall accuracy accounting for the balance.

Not Class Balanced

91.6% accuracy

Accuracy for class 1: 0.9666666666666667

Accuracy for class 2: 0.7

Accuracy for class 3: 0.88

Accuracy for class 4: 0.95

Accuracy for class 5: 0.9714285714285714

Accuracy for class 6: 0.875

Accuracy for class 7: 1.0

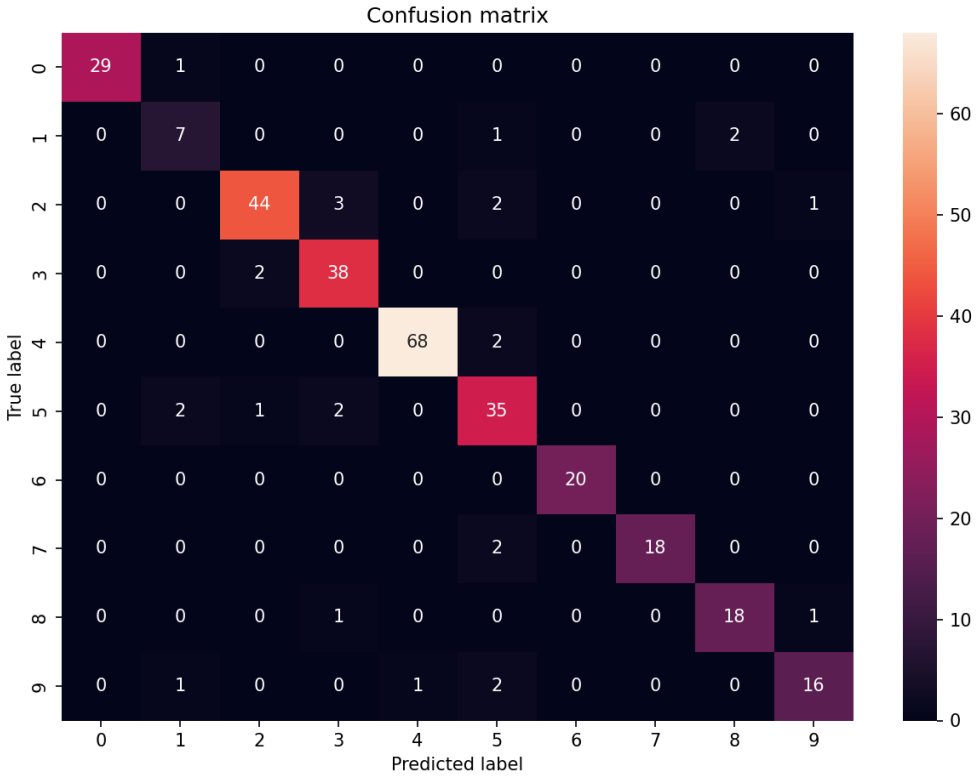
Accuracy for class 8: 0.9

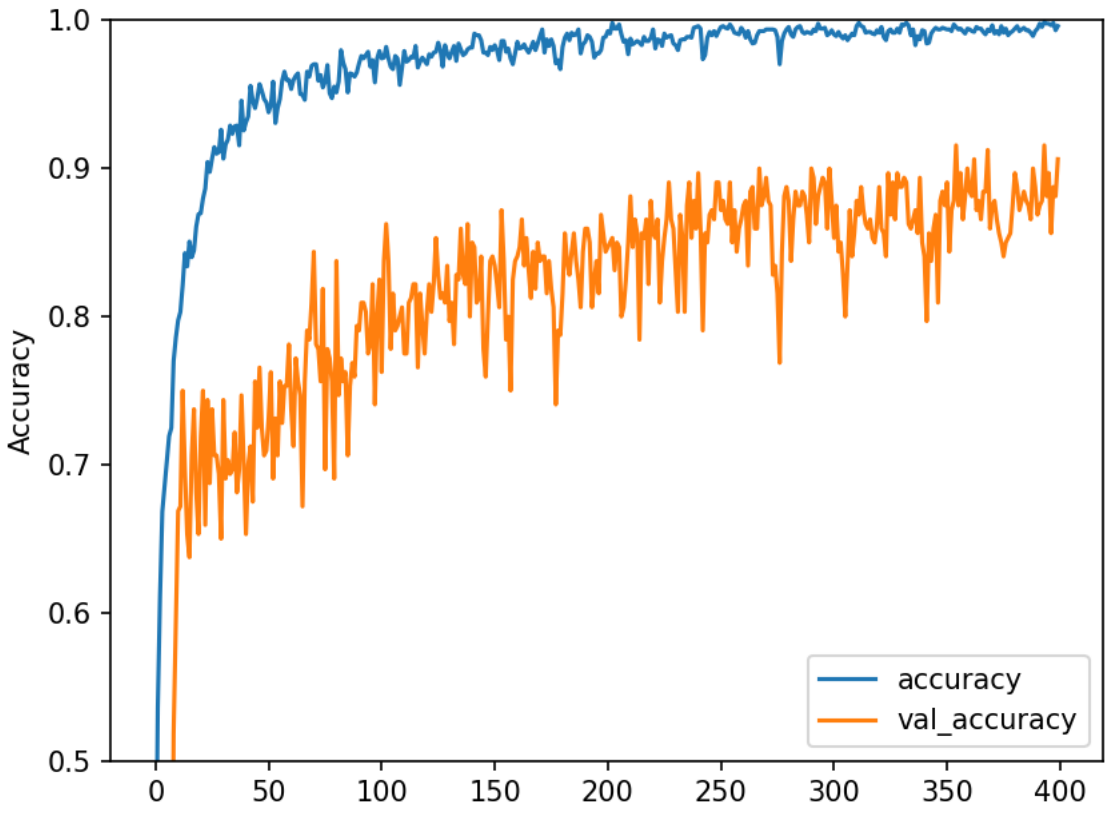
Accuracy for class 9: 0.9

Accuracy for class 10: 0.8

Average accuracy: 0.8943095238095239

With the confusion matrix and accuracy through the epochs below





Model accounting for class balance

91.6% accuracy

Accuracy for class 1: 1.0

Accuracy for class 2: 0.7

Accuracy for class 3: 0.92

Accuracy for class 4: 0.9

Accuracy for class 5: 0.9714285714285714

Accuracy for class 6: 0.825

Accuracy for class 7: 1.0

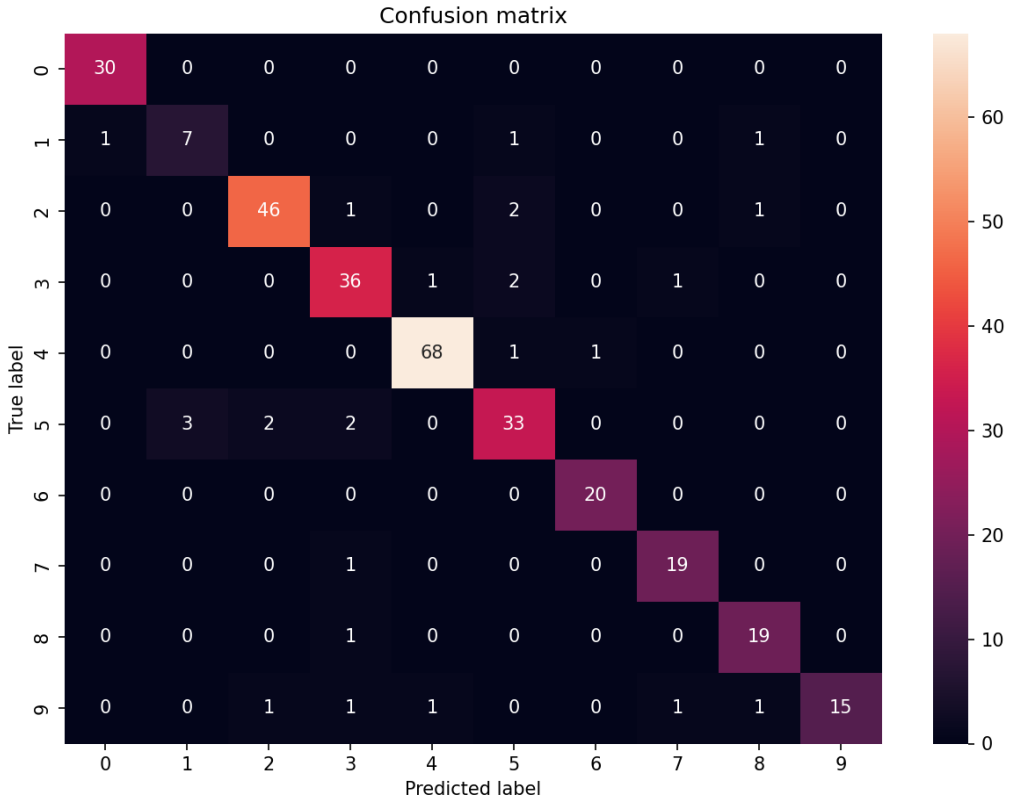
Accuracy for class 8: 0.95

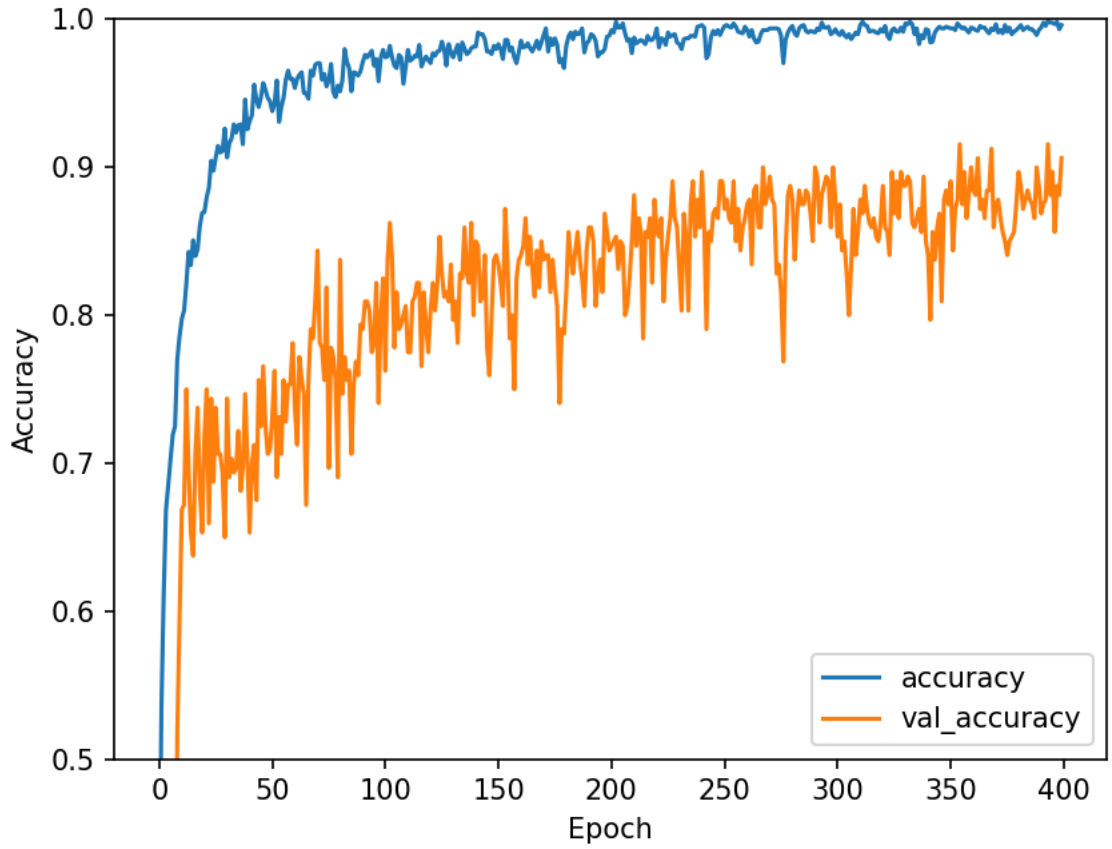
Accuracy for class 9: 0.95

Accuracy for class 10: 0.75

Average accuracy: 0.8966428571428571

With a confusion matrix and accuracy through epochs shown below

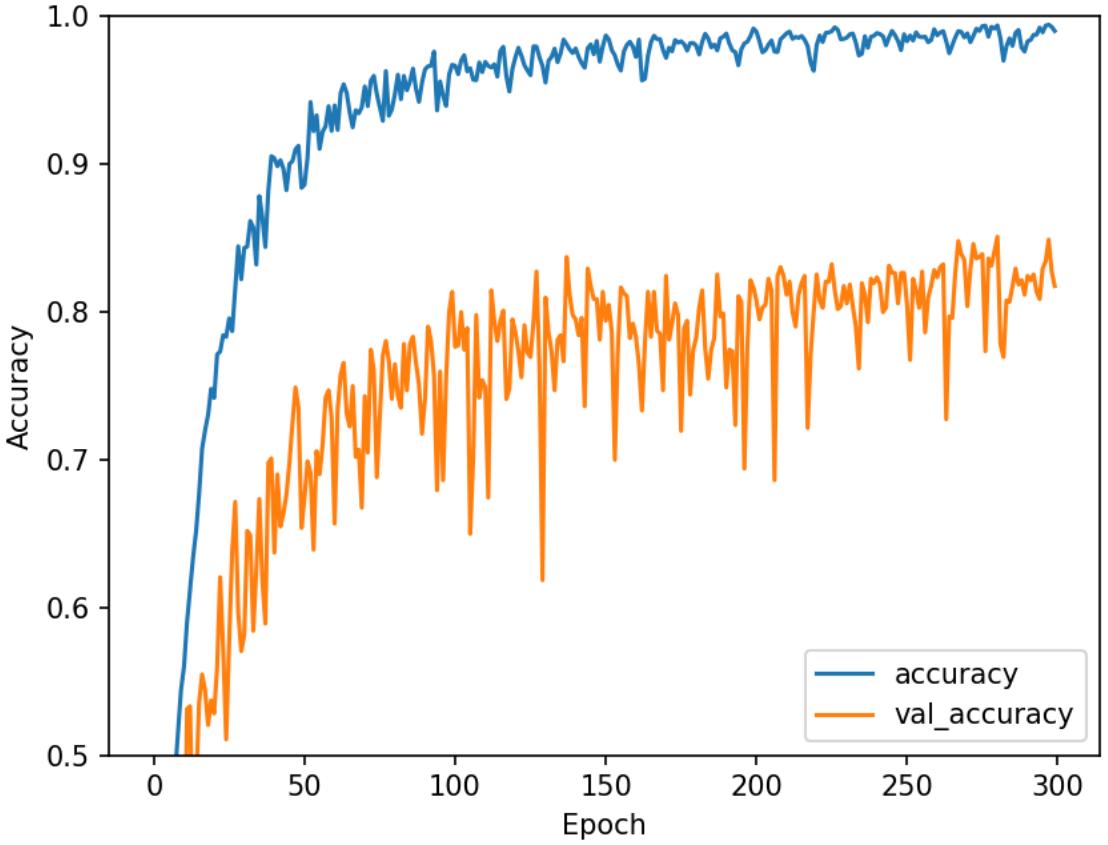




Overall, the models perform similarly with similar patterns in growth rate, the balanced class however performs slightly above the non-balanced class (89.6 vs 89.4) but that could be due to variability. As to why the non-balanced model performed so well on the balanced dataset, I suspect that the data augmentation really helps with that as the augmentation will create new images based on the existing images allowing for extra training data. But as the Balanced model did perform better, I will be using that architecture for the fine dataset.

Fine dataset

Using the Fine dataset and training for 300 epochs using VGG16 Architectural style with data augmentation, batch normalization, callback to the best performing model and accounting for class imbalance the fine dataset came out to an accuracy of 85.1% for both overall test accuracy and test accuracy accounting for class imbalance on test data. Though there still were some classes that performed poorly showing that class imbalance still exist. Overall most of the classes performed quite well. 300 epochs were chosen for time constraints but as the validation accuracy has yet to plummet/overtrain we can assume that with more epochs, performance probably would have been better. Overall it seems like the performance was transferred over from the course dataset to the fine dataset quite well as the fine dataset performed fairly well for being over 10x the number of classes and only a 5% drop in performance. Below are some graphs and images about the model's training.



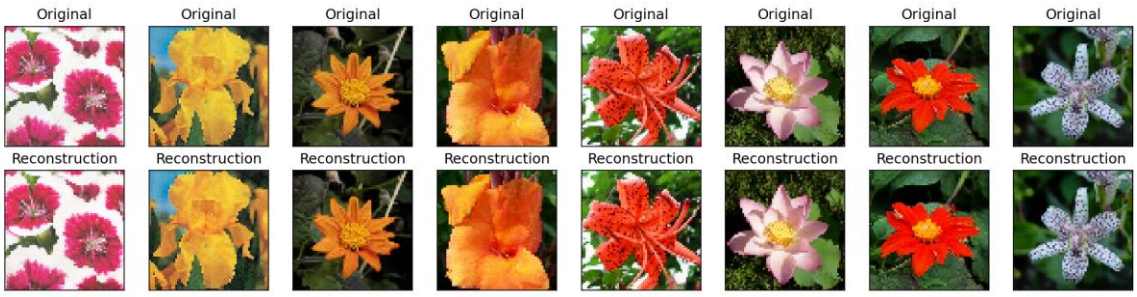
As we can see, the Val accuracy growth does seem to slow down significantly after the first 150 epochs, but the variability does seem to get better over time and the best test accuracy was hit towards the end. Potentially meaning that higher highs can be hit with more epochs, but unfortunately due to time constraints, I won't explore that. Overall, the model performs quite well

above as out of the 16 it gets 2 wrong which matches the accuracy score of 80% and with the ones it does predict wrong it is quite understandable as for example picture predicts balloon flower. The flower does indeed look quite like a balloon flower with a similar colour. See Balloon flower below.

Task 2A

Auto encoder

To start the Diffusion model, I have followed the Unet Architecture. As it is known to be good at this sort of tasks. However, the Unet Architecture was a bit big for my device to handle so I have scaled it down a bit. Along with dropping the pixel size down to 64*64 pixels. Below are the results from the autoencoder.



Mean error: 0.0007872945316040578

Standard deviation of error: 0.0020087447531746665

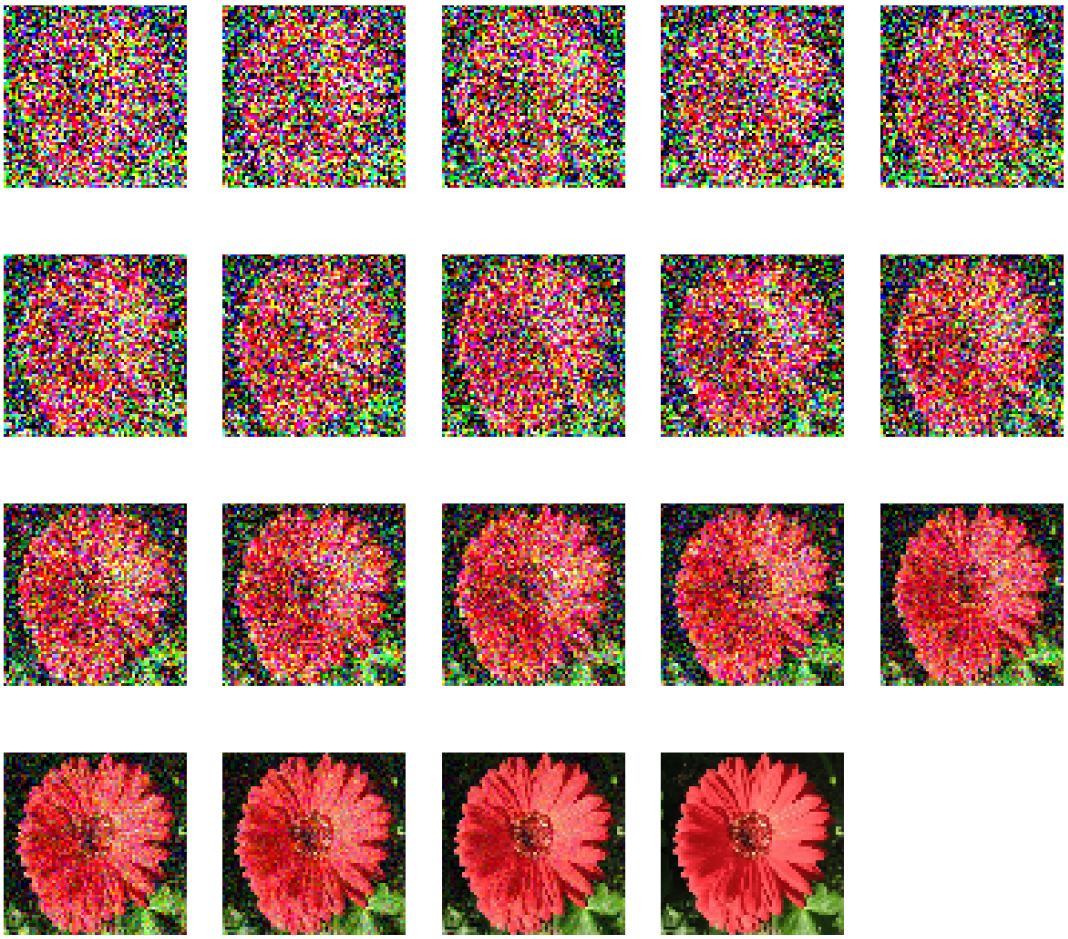
As seen above visually, this architecture works quite well. The mean error and standard deviation of the pixel error are also very well. Signalling that the minimized UNet model works quite well and can be used for part b of this task for the diffusion.

Task 2B

Task 2b requires us to create an image from noise. This is done by breaking down a train image into multiple images each with different amounts of noise added to them. I then paired these images up with the input being the noisier image and the output being the less noisy image. As seen below.

x Images index 0 – 19

y Images 0 - 19

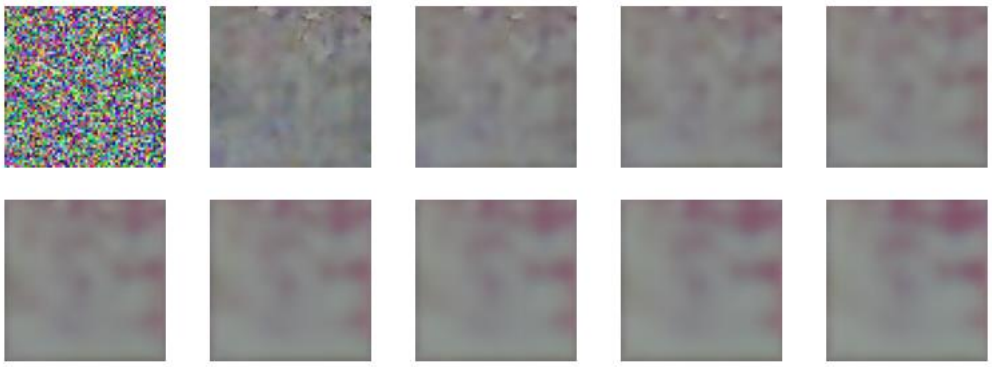


In the above images, the x[0] image is the image with full noise added to it and y[0] will be with 0.95x noise level. Then x[1] will be used as 0.95 and y[1] will be 0.9x noise level. This is done until we get to the no noise images and then we do it for the next image as seen above. To do this I generated the noise by using a gaussian distribution cantered around 0 and the standard deviation between 0 and 1. As we get more noisy images, we increase the standard deviation by 1/number of generated images per image. The model is then trained on this data with each x_train image pointing to the slightly less noisy version stored in y_train.

I trained 3 models with the above dataset. Once with each image being broken down into 20 images and once with each image being broken down into 15 images and lastly 10 images.

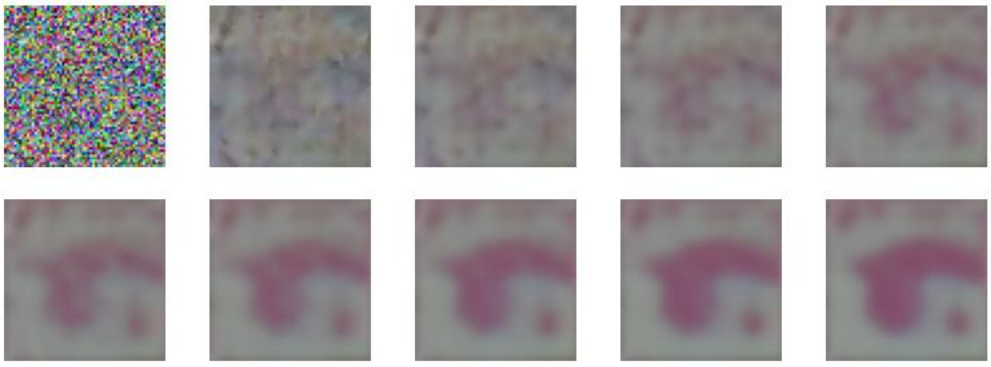
Model with each image broken down into 10 images.

Random Noise



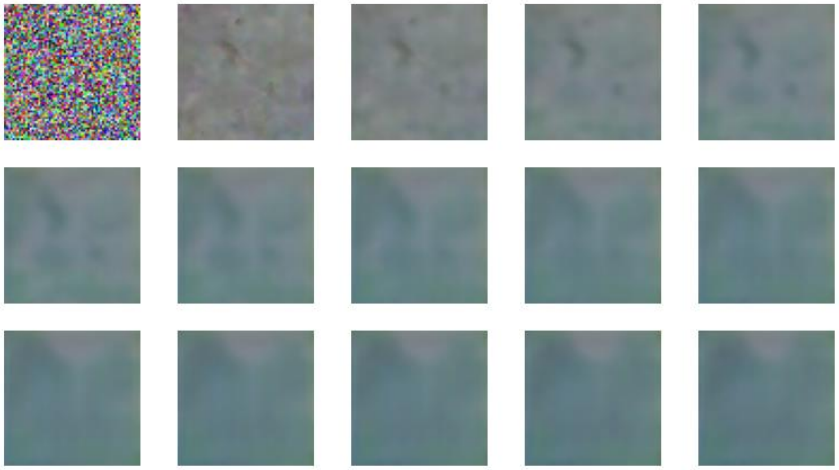
Gaussian Distribution SD = 0.5

Gaussian Distribution SD = 0.3



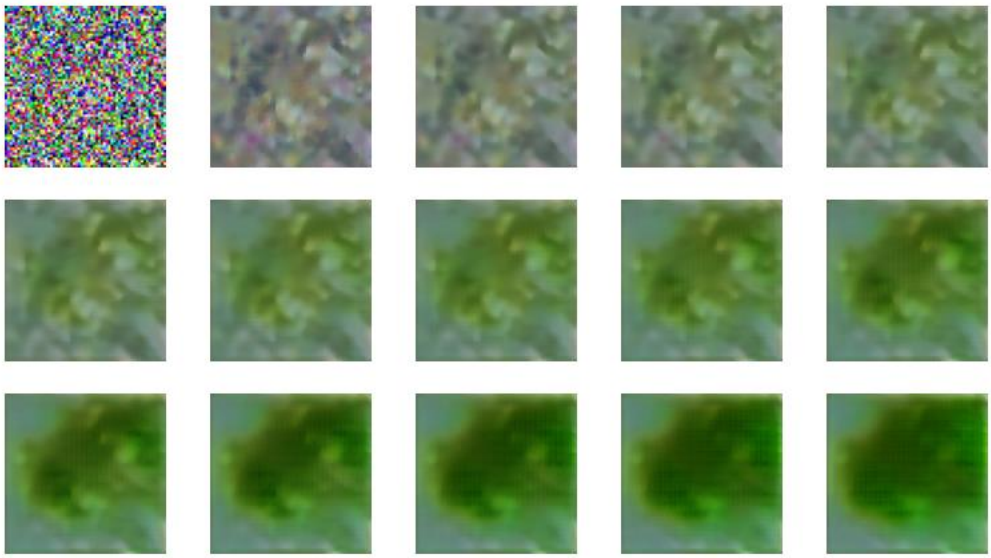
Model with each image broken down into 15 images.

Random noise image



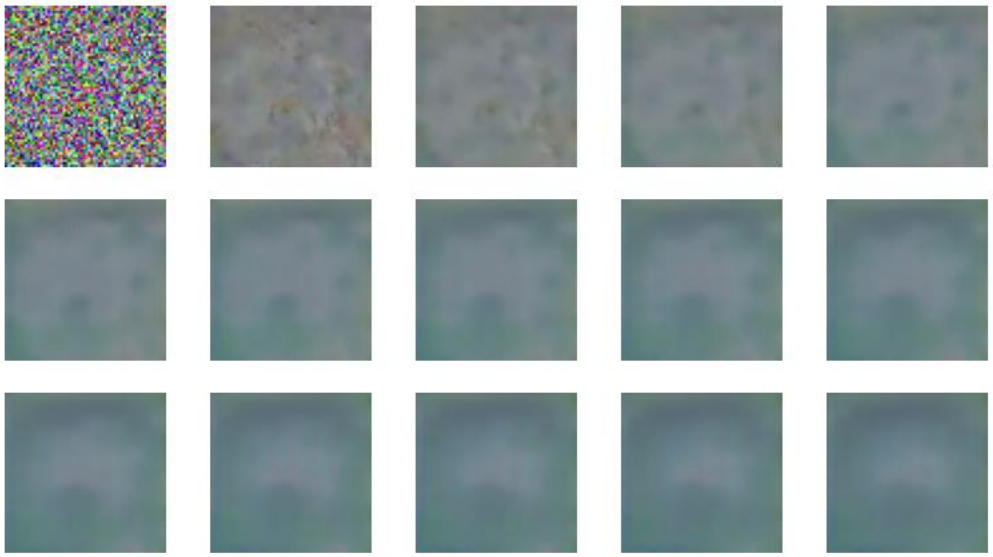
Notes white cone shape at the top

Gaussian Distribution Image standard deviation = 0.5



Note: Looks like broccoli with white spots?

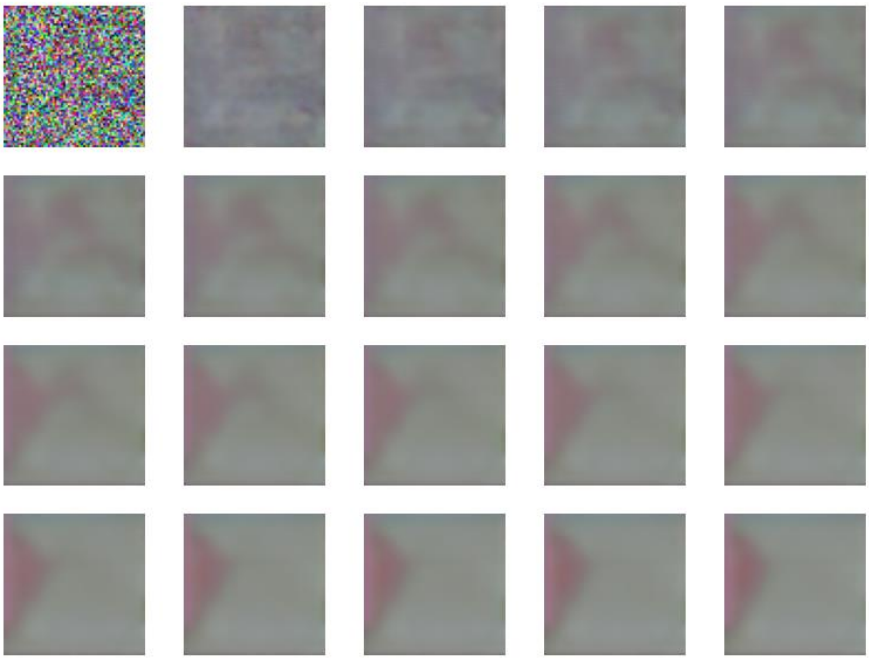
Gaussian Distribution Image standard deviation = 0.3



Notes Looks like White Australia in blue ocean

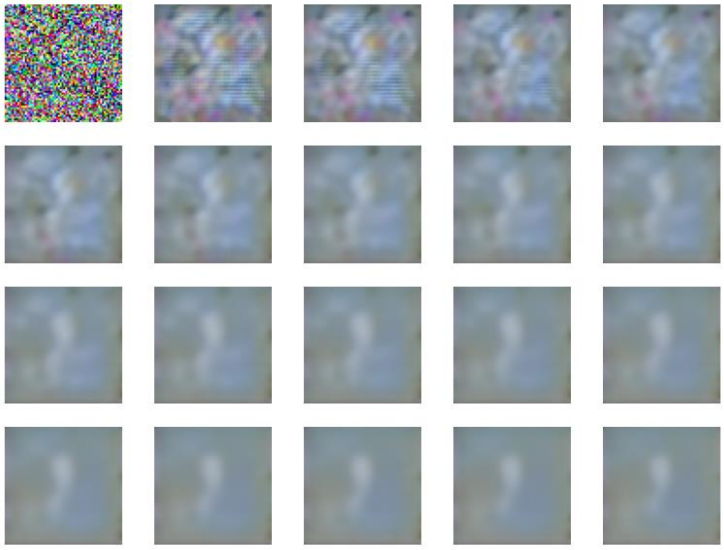
Model with each image broken down into 20 images.

Random Noise Image



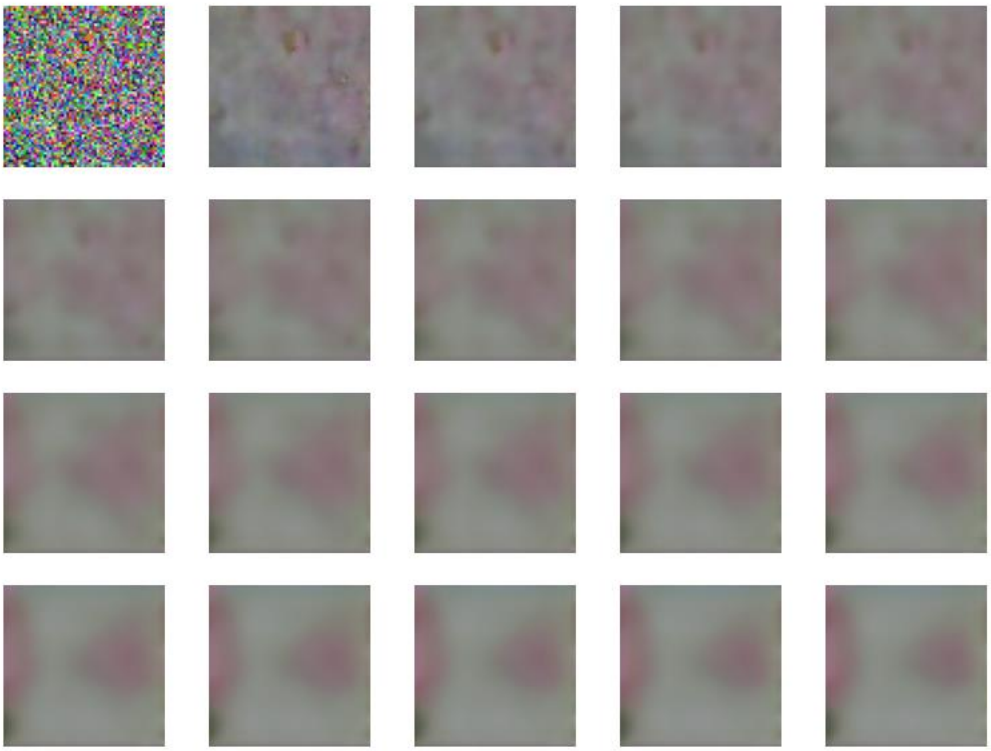
Note Red/Pink cone shape at the left

Gaussian Distribution Image standard deviation = 0.5



Note: blue blob with a white thing in the middle.

Gaussian Distribution Image standard deviation = 0.3



Note 2 pink blobs

Unfortunately, I couldn't get past the stages of colourful blobs. The images generated could be said that there were some resemblances of flowers. Such as the cone shape and the stalk that can be potentially seen in the earlier images. But something that was interesting was how similar the random generated images were to the 0.3 standard deviation gaussian distribution for all image models. Along with how different the random noise image and the 0.5 standard deviation gaussian distribution. As it seems like there is no overlap with colours and shapes except for 10 image model 0.5 sd. But the 0.3 standard deviation not only has the same colour but a similar shape. When rerun multiple times with the same methods, the images tend do come out quite similarly. As for the 3 different models it can be noted that the 10 image one only kept the colour pink whilst the other 2 had different colours and patterns based on noise.