COSC420: Assignment 1 report

Junyi Shen (8386129)
April 13, 2025

1 Introduction

In this assignment, I implemented a convolutional neural network(CNN) to classify images on the Oxford flowers dataset. I also implemented an auto-encoder that can encode an image into latent space and decode it back to the original image. I developed a latent diffusion de-noising model (in UNet architecture) that can de-noise noisy images in the latent space and eventually generate different types of flower images from random noise over 10 de-noising steps. I will explain the architecture of my models, the choice of hyper-parameters and the reason I chose them, as well as the visualized test results. Finally, I will share some problems I encountered during the development process and how I solved these problems.

2 Task 1 Convolutional Neural Networks

In this task, I implemented a CNN model that is capable of classifying images from Oxford flower dataset. My model achieves 81% accuracy on the coarse-grained dataset and 72% accuracy on the fine-grained dataset. The confusion matrices will be shown in the test results section (Figure 1 and Figure 2).

2.1 Dataset

The Oxford flower dataset includes a coarse-grained set and a fine-grained set. The coarse-grained version has 10 different labels and the fine-grained version has 102 different labels.

Dataset	Training data count	Validation data count	Test data count	Total labels	Max label count (training set)	Min label count (training set)
coarse	2322	390	390	10	798	49
fine	6149	1020	1020	102	231	20

Table 1: Oxford flowers dataset overview

Overall, the dataset is small and highly imbalanced. In the training data, the most common label in coarse-grained version appears 798 times and the least common one appears only 49 times. The most common label in fine-grained version appears 231 times while the least common label appears only 20 times.

The main challenge for the classification model is that it is very easy to get overfitting (The model captures too much detailed information in the training set and

does not generalise well in the validation and test dataset). In my experimentation, in the coarse-grained set, the training accuracy is very easy to reach 99%, but the validation and test accuracy is difficult to reach 80%. Many approaches should be used to help the model generalise as well as possible, including limiting the complexity of the architecture, data augmentation, adjusting batch size, and regularisation methods.

2.2 Image size

The image size should be between 32×32 and 500×500. The image size I pick is 96×96. The reason for picking this size is that a small size (like 32×32 or 64×64) lacks enough information for the model to capture, especially in the fine-grained version, while images with a large size (larger than 128×128) contain too many details so the model will easily capture too many details in the training data and would not generalise well, especially when the dataset is small, like the Oxford flower dataset.

2.3 Architecture

Table 2 shows the architecture of my CNN model. It lets the input data go through five convolutional layers, then the data is flattened and goes through two fully-connected layers. The number of neurons in the output layer is the same as the number of labels (10 for coarse-grained and 102 for fine-grained).

The number of parameters of my CNN model is 760074 for the coarse-grained version and 807270 for fine-grained version, both less than 1 million. The reason why I use 5 fully-connected layer each followed by a max pooling is that I need to reduce the image size to very small (3×3 in my architecture) so the final convolutional layer, which has 3×3 kernels, can capture features of the entire image.

I tried to increase the number of neurons, and add more convolutional layers, but didn't see any significant improvements in terms of test accuracy, possibly because the more complex the architecture is, more prone it is to capture too many subtle details and overfit to training data.

As Table 3 shows, I tried to add the number of neurons in each convolutional layer, even though the test accuracy on the coarse-grained dataset increases from 81% to 83%, the test accuracy on the fine-grained dataset decreases from 72% to 66%. This is probably because more complex architectures are more prone to capture too many details in the training set and do not generalise well.

I also tried to add a softmax layer as the last layer of the neural network, but I get worse results (coarse-grained test accuracy decreases from 81% to 78%).

2.3.1 Data augmentation

Data augmentation is used to deal with data imbalance issues to help the model generalise well on each label. My approach of data augmentation is to augment the training

Layer type	Filters	Kernel size	Stride	Padding	Output Size (number of neurons)
Input					96×96 × 3
Convolutional layer	16	3	1	1	96×96 × 16
Max pooling		2	2		48×48 × 16
Convolutional layer	32	3	1	1	48×48 × 32
Max pooling		2	2		24×24 × 32
Convolutional layer	64	3	1	1	24×24 × 64
Max pooling		2	2		12×12 × 64
Convolutional layer	128	3	1	1	12×12 × 128
Max pooling		2	2		6×6 × 128
Convolutional layer	256	3	1	1	6×6 × 256
Max pooling		2	2		3×3 × 256
Fully-connected layer					128
Fully-connected layer					512
Fully-connected layer					10 (coarse-grained) or 102 (fine-grained)

Table 2: CNN model architecture

Architecture	Dataset	Epoch	Training accuracy	Validation accuracy	Test Accuracy
ConvLayers 16, 32, 64, 128, 256	coarse	50	100%	81%	81%
	fine	50	99%	72%	72%
ConvLayers 32, 64, 128, 256, 512	coarse	50	100%	81%	83%
	fine	50	100%	67%	66%

Table 3: Performance comparison between different architectures

dataset and let all the labels have the same number of images. For example, in the coarse-grained version, the most common label appears 798 times in the training set. I augment all the other labels and let each label has 798 images. By this means, I ensure that each label has the same number of training data, so the model would not be biased toward the most common label.

The ways of augmentation include random horizontal flip, random rotation, random affine, and color jitter. After augmentation, I combine the newly generated images and the original images to create an augmented training set. As shown in Table 4, the test accuracy on coarse-grained test dataset increases significantly from 66% to 81% after enabling data augmentation.

	Dataset	Epoch	Training accuracy	Validation accuracy	Test Accuracy
Disable data augmentation	coarse	50	99%	67%	66%
Enable data augmentation	coarse	50	100%	81%	81%

Table 4: Accuracy comparison between before and after performing data augmentation

2.4 Regularisation

Batch normalisation, dropout, and weight decay are three most common methods for regularisation to reduce overfitting. Batch normalisation normalizes the output of a layer and increases stability during training. Dropout randomly deactivates some neurons to avoid some neurons from updating much more than others. Weight decay adds a penalty to the loss function to prevent some weights from being too large.

I experimented with all three methods, tried different hyper-parameters, and tested

Dataset	Epoch	Batch Norm	Weight decay	Dropout	Training accuracy	Validation accuracy	Test Accuracy
coarse	50	N/A	N/A	N/A	99%	63%	66%
coarse	50	N/A	0.001	N/A	99%	63%	66%
coarse	50	enabled	N/A	N/A	100%	78%	80%
coarse	50	enabled	0.001	N/A	100%	81%	81%
coarse	50	enabled	0.001	0.3	99%	77%	76%
coarse	50	enabled	0.0005	N/A	100%	76%	78%
coarse	50	enabled	0.005	N/A	100%	77%	81%

Table 5: Comparison of accuracy between multiple regularisation choices

the results. As shown in Table 5, enabling batch normalisation and setting a weight decay to 0.001 perform best for my model. Either decreasing or increasing the weight decay doesn't see performance improvements, and adding dropout sees an accuracy drop.

2.5 Other hyper-parameters

2.5.1 Batch size

For batch size, I tried 4, 16, 64, 128, and 256 and found that a batch size of 128 can perform better than the other sizes.

2.5.2 Learning rate

The learning rate with the best performance for my model is 0.001. I tried to decrease the learning rate to 0.0005 (decrease from 81% accuracy to 78%) or increase it to 0.005(the same accuracy as 0.001), but both didn't see a performance increase.

2.5.3 Optimizer

My choice of optimizer is Adam, which combines the benefit of SGD with momentum (smooth out gradients) and RMSProp (adjust learning rate based on gradients).

2.6 Test results

My CNN model can achieve 81% accuracy on the coarse-grained test dataset and 72% on the fine-grained dataset. I also generated the confusion matrixes of both test datasets.

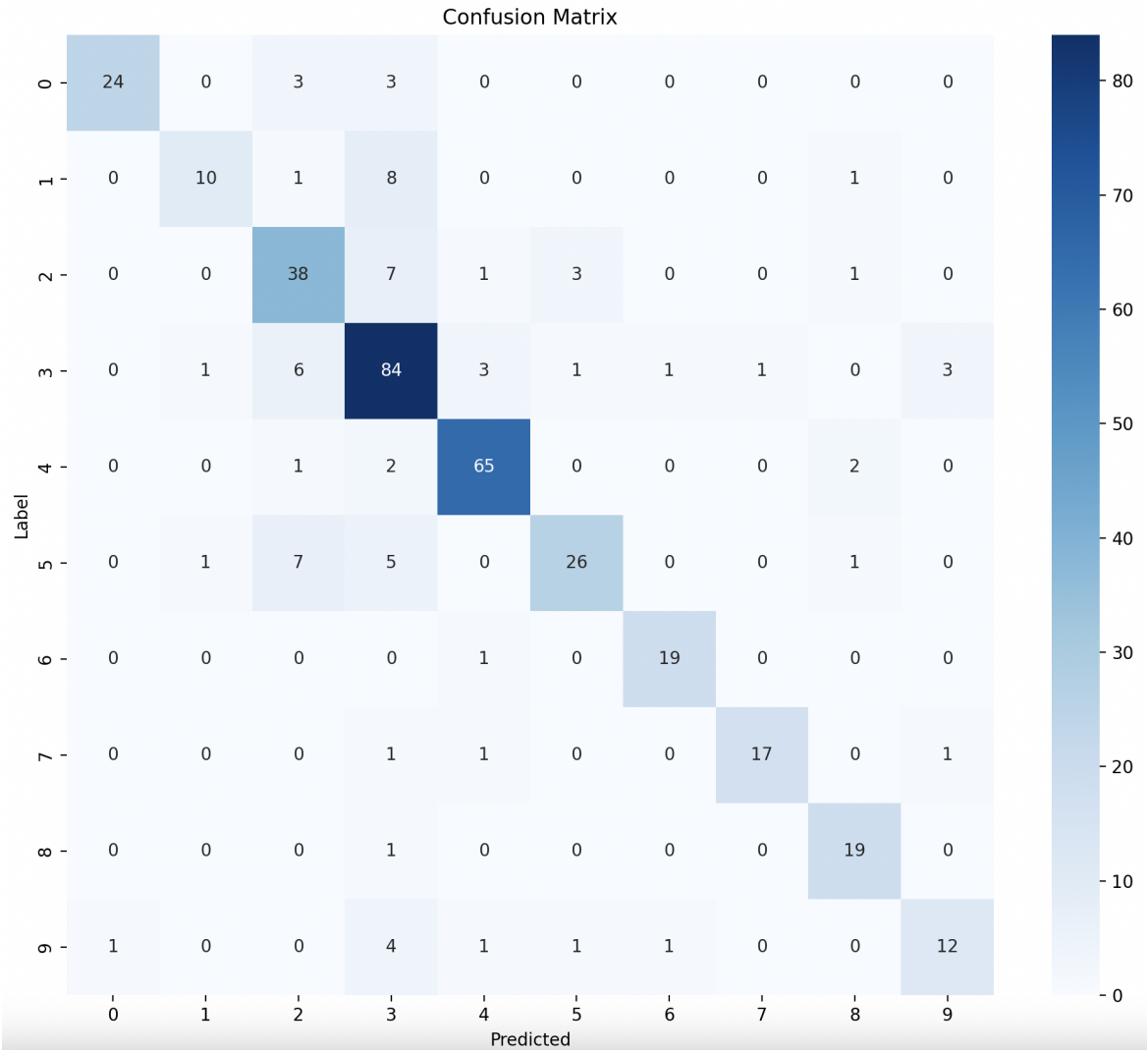

Figure 1: Confusion matrix on Oxford flowers coarse-grained test dataset

As shown in the confusion matrixes (Figure 1 and Figure 2), the model generalises well across all kinds of flowers in both coarse-grained and fine-grained datasets, except for labels 2 and 31 in the fine-grained test dataset which the model finds hard to classify.

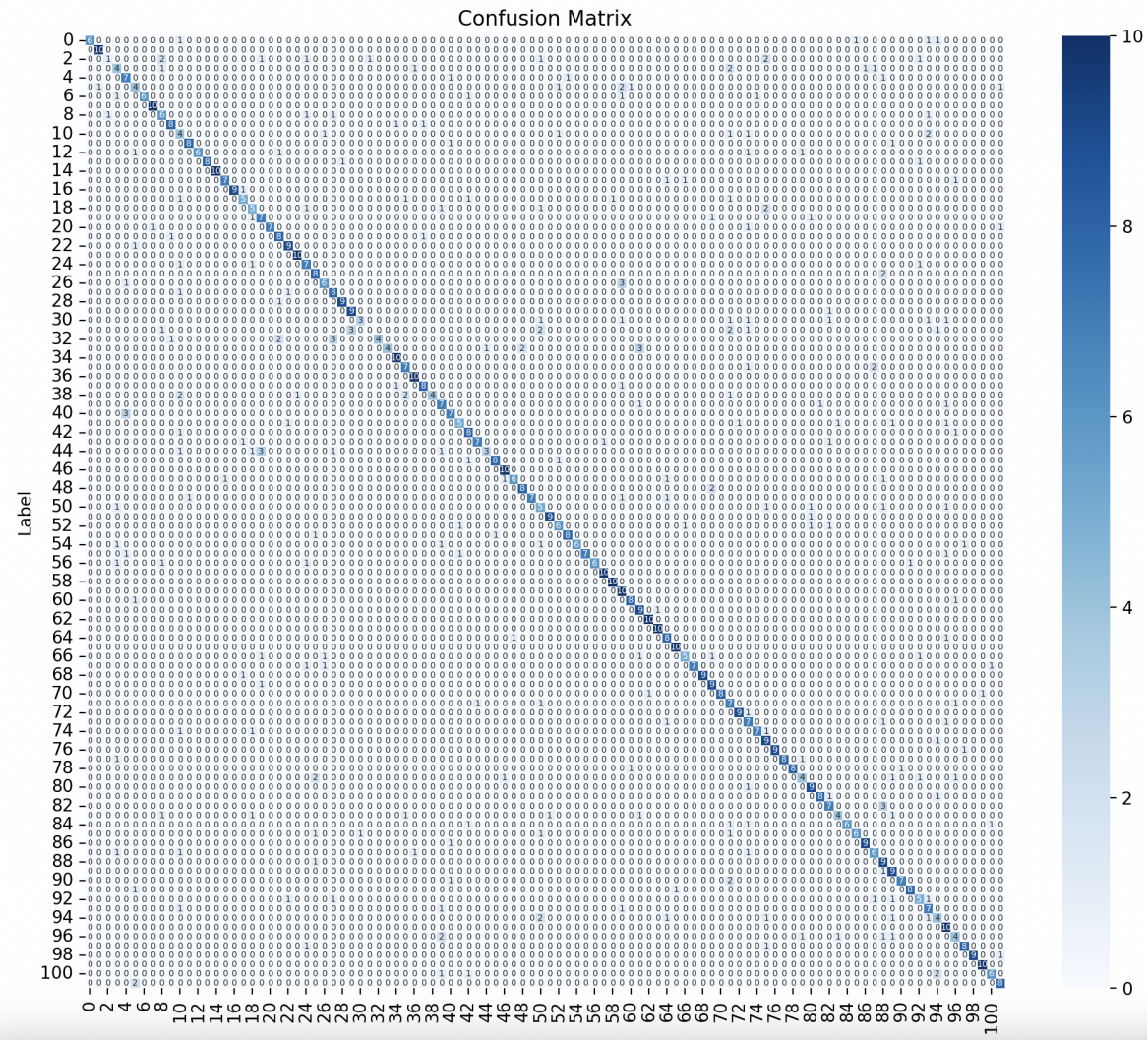

Figure 2: Confusion matrix on Oxford flowers fine-grained test dataset

3 Task 2a Auto-encoder

The auto-encoder consists of two parts: the encoder and the decoder. The task of the encoder is to encode an image into a latent space, and the task of the decoder is to decode the latent image back into the original image. The size of the images chosen for task 2 is the same as task 1 (96×96).

3.1 Architecture

The architecture of my encoder is three convolutional layers, with 16, 32, and 64 filters, respectively. Each of the convolutional layers has a kernel of 3 to capture features, and a stride of 2, enabling it to reduce the image width and height into half. The size of the original input image is $96 \times 96 \times 3$, where 96 is the image width and height and 3 is the colour channels(RGB), and after going through the three convolutional layers, its size in the latent space will become $12 \times 12 \times 64$, which is one third the original size.

The architecture of my decoder model is three transpose convolutional layers, with

Layer type	In channels	Out channels	Kernel size	Stride	Padding	Output Size (number of neurons)
Input						96×96 × 3
Convolutional layer	3	16	3	2	1	48×48 × 16
Convolutional layer	16	32	3	2	1	24×24 × 32
Convolutional layer	32	64	3	2	1	12×12 × 64

Table 6: Encoder architecture

64, 32, and 16 filters, respectively. Each transpose convolutional layer has a stride of two, allowing it to double the input width and height. After going through the three transpose convolutional layers, the $12 \times 12 \times 64$ latent image will return to $96 \times 96 \times 3$, which is the size of the original image.

The auto-encoder is trained on the Oxford flower coarse-grained training dataset

Layer type	In channels	Out channels	Kernel size	Stride	Padding	Output Size (number of neurons)
Input						12×12 × 64
Transpose Convolutional layer	64	32	3	2	1	24×24 × 32
Transpose Convolutional layer	32	16	3	2	1	48×48 × 16
Transpose Convolutional layer	16	3	3	2	1	96×96 × 3

Table 7: Decoder architecture

over 150 epochs. The loss function is Mean Square Error(MSE) because it needs to compare the difference of each pixel between the original and reconstructed images. The image size is also 96×96 and the batch size is chosen to be 16. The learning rate is 0.0001. These general hyper-parameters can help train the model well and reconstruct clear images.

3.2 Test results

Figure 3 shows ten image examples and their reconstructed image. All ten image samples are chosen from the coarse-grained test dataset, and each of them has a different label. As the reconstructed images show, the auto-encoder can reconstruct clear images, although a little bit blurry due to reconstruction loss.

Table 8 shows the mean and standard deviation of error when comparing the ten reconstructed images with the original images shown in Figure 3. It also shows the overall error mean and error standard deviation of all 10 test images.

Figure 3: Original images and reconstructed images (first row is the original images, and the second row is the reconstructed images) Test images source: coarse-grained test dataset, image indices 0, 10, 20, 30, 50, 60, 110, 120, 210, 260. All of them have different labels

Image index	Image label	Error mean	Error standard deviation
0	0	0.0265	0.0371
10	1	0.0260	0.0274
20	4	0.0209	0.0249
30	2	0.0217	0.0263
50	3	0.0373	0.0340
60	5	0.0225	0.0258
110	7	0.0302	0.0321
120	9	0.0450	0.0393
210	6	0.0275	0.0279
260	8	0.0290	0.0284
Error of all 10 images		0.0287	0.0315

Table 8: Error mean and standard deviation of 10 test images of different labels from coarse-grained test datasets

4 Task2b UNet De-noising Model

My latent denoising model is trained to remove noise gradually from images in latent space. Starting from pure random noise, it is able to generate clearer and clearer images over 10 de-noising steps. It can generate different images from different random noises.

4.1 Training process

In the training process, for each training image sample, 10 noisy images with different levels of noise are generated, from a bit of noise to almost pure noise. The noisy image of the next level is generated by adding some noise to the image of the previous level. After these noisy images are generated, they are encoded by the encoder to the latent space. The input for the model is a more noisy latent image, and the label is the latent image of the previous noisy level. By this means, the model is trained to gradually remove noise from images in the latent space. As shown in Figure 4, each training image is added gradual noise like this example in 10 de-noising steps. Each de-noising step is based on the previous de-noising step and adds some new noise to it.

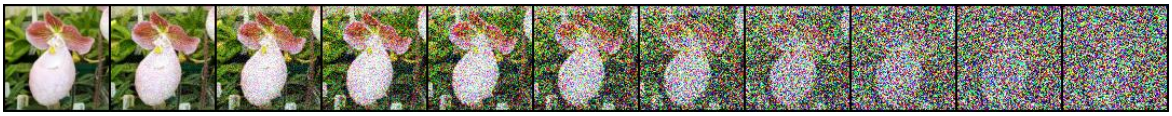

Figure 4: Sample training data for the UNet de-noising model

During training, the model is fed in the next de-noising step and expected to output the previous de-noising step. Then mean square error(MSE) is calculated and model weights are updated after each batch.

When generating images from random noise, the random noise is encoded to the latent space, and the model is able to gradually remove noise and generate clearer and clearer images in the latent space. When going through the denoiser model multiple times, different random noises will generate different images because the model is trained on a variety of images.

4.2 Architecture

As shown in Table 9, my UNet de-noising model receives an input of $12 \times 12 \times 64$, which is the output of the encoder (converts $96 \times 96 \times 3$ to $12 \times 12 \times 64$). The input goes through multiple convolutional layers and downsamples with two max pooling layers. After going through multiple bottleneck layers, the data upsamples back to $12 \times 12 \times 64$ with multiple convolutional layers and transpose convolutional layers. The input of layer *up2_3* is 256+256, this is because this layer receives the outputs from both layer *up2_2* and *down2_3*. Similarly, the input of layer *up1_3* is 128+128, which is the concatenation of the outputs of layer *up1_2* and *down1_3*. These skip connections bring back the lost details during downsampling and help the model generate clearer images.

4.3 Hyper parameters

4.3.1 Learning rate

I set a learning rate to 0.0001 for most of my development process. I also tried 0.0005 and 0.00005 but didn't get better results. So I used 0.0001 for the final version.

4.3.2 Epochs

I train 100 epochs each time and check the results (see if the loss is still decreasing and generated images are getting better). If the results get better, I continue training from the existing model. In the first 300 epochs, the generated images are very blurry and unrecognizable (Figure 5). But it gets much better after running for 500 epochs (Figure 6). Every 100 epochs, the loss decreases, even reaching 1000 epochs (although decreasing very slowly). So, it is possible to continue training the model, although I just train it for 1000 epochs. The test results of training 1000 epochs are shown in Figure 7.

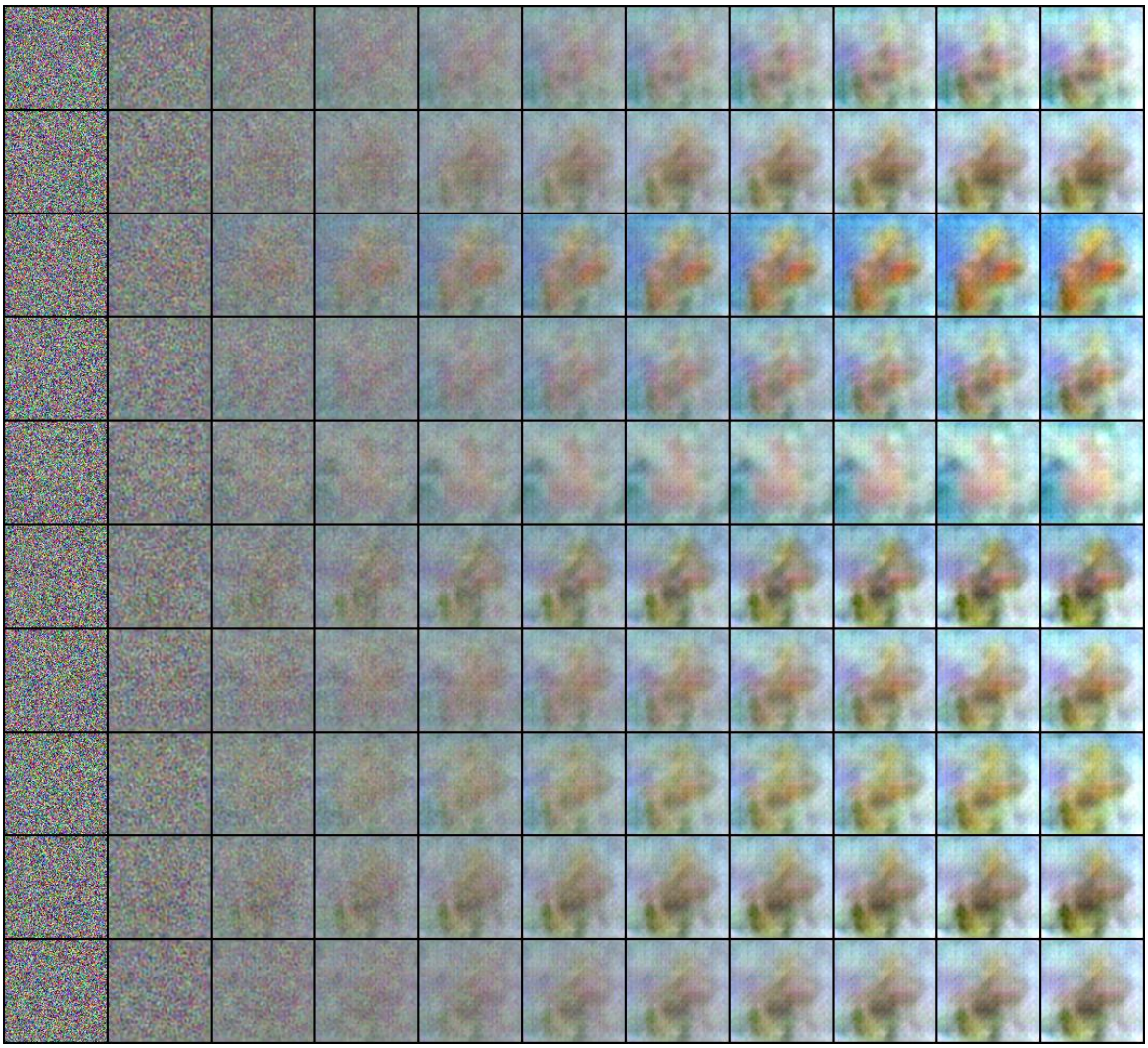

Figure 5: Test results after training for 300 epochs

Figure 6: Test results after training for 500 epochs

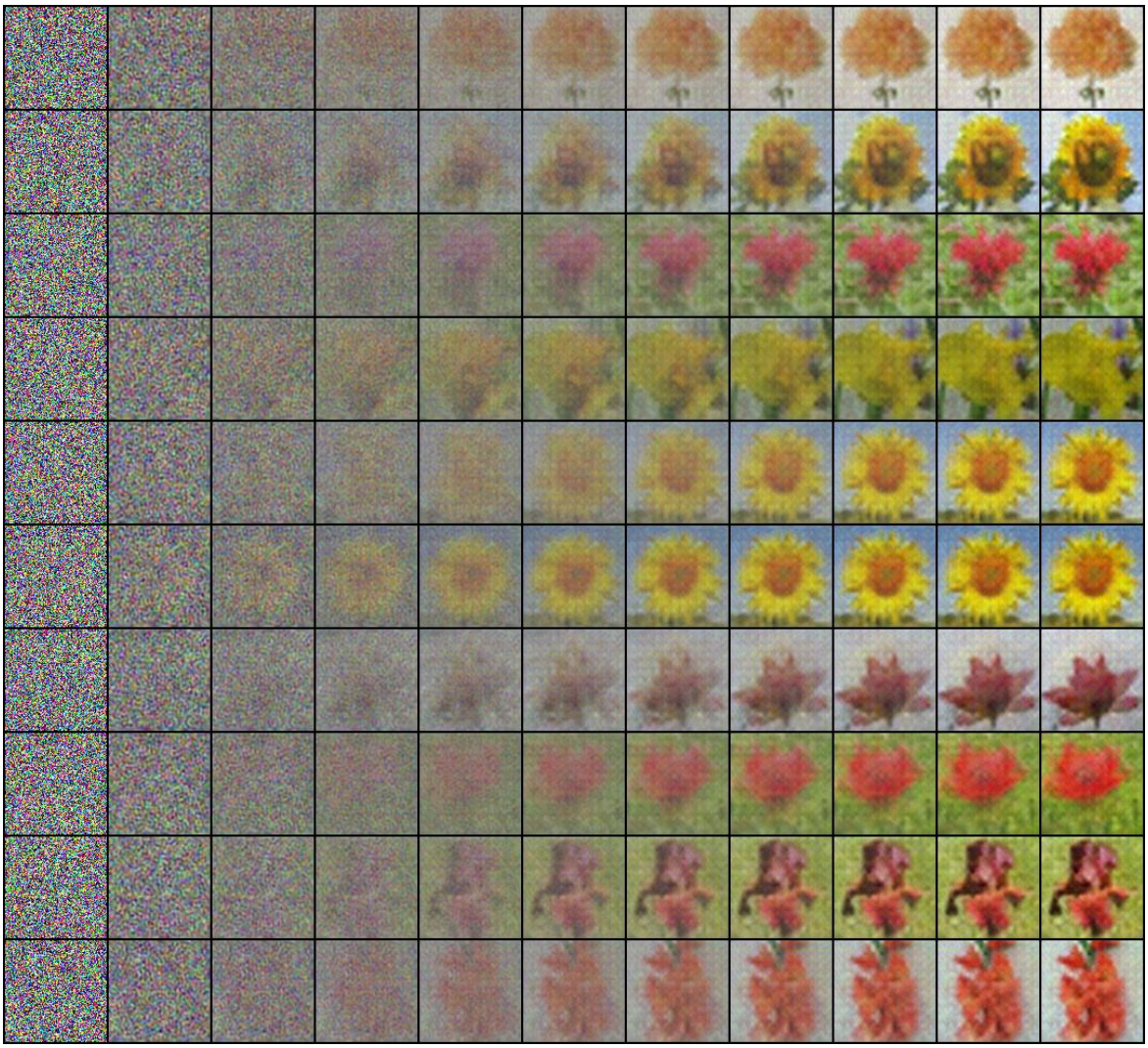

Figure 7: Test results after training for 1000 epochs

Layer name	Layer type	In channels	Out channels	Kernel size	Stride	Padding	Output Size (number of neurons)
input							12×12 × 64
down1_1	Convolutional layer	64	128	3	1	1	12×12 × 128
down1_2	Convolutional layer	128	128	3	1	1	12×12 × 128
down1_3	Convolutional layer	128	128	3	1	1	12×12 × 128
pool1	Max pooling			2	2		6×6 × 128
down2_1	Convolutional layer	128	256	3	1	1	6×6 × 256
down2_2	Convolutional layer	256	256	3	1	1	6×6 × 256
down2_3	Convolutional layer	256	256	3	1	1	6×6 × 256
pool2	Max pooling			2	2		3×3 × 256
bottlenect1	Convolutional layer	256	512	3	1	1	3×3 × 512
bottlenect2	Convolutional layer	512	1024	3	1	1	3×3 × 1024
bottlenect3	Convolutional layer	1024	512	3	1	1	3×3 × 512
bottlenect4	Convolutional layer	512	256	3	1	1	3×3 × 256
up2_1	Transpose Convolutional layer	256	256	3	2	1	6×6 × 256
up2_2	Convolutional layer	256	256	3	1	1	6×6 × 256
up2_3	Convolutional layer	256+256	128	3	1	1	6×6 × 128
up1_1	Transpose Convolutional layer	128	128	3	2	1	12×12 × 128
up1_2	Convolutional layer	128	128	3	1	1	12×12 × 128
up1_3	Convolutional layer	128+128	64	3	1	1	12×12 × 64
output	Convolutional layer	64	64	3	1	1	12×12 × 64

Table 9: Architecture of UNet de-noising model

4.3.3 Batch size

In a batch, for each image sample, the model will train 10 times (one for a de-noising step), so the batch size should not be too large. I set the batch size to 16, so in each

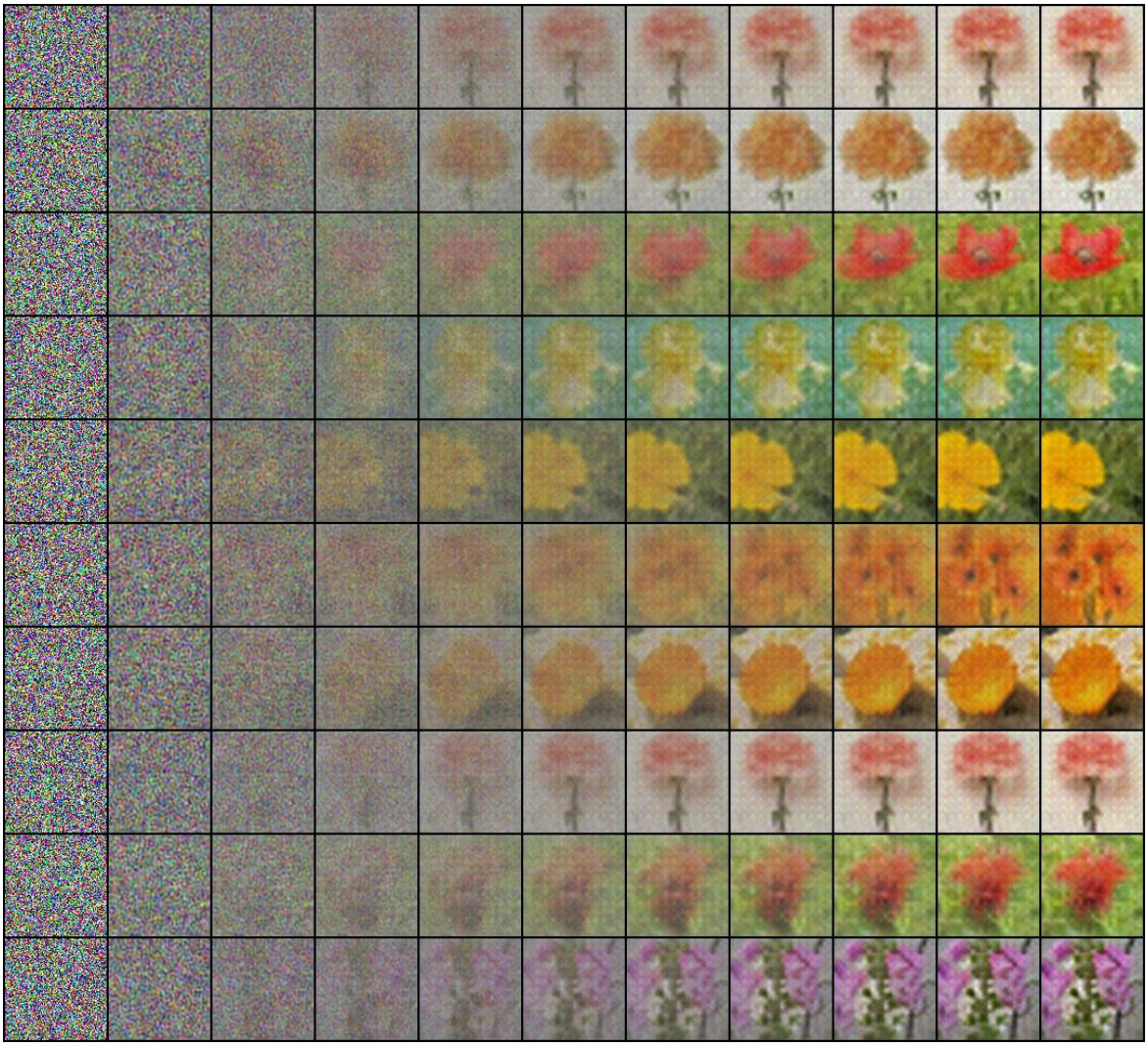

Figure 8: The result of my final model (15.9 million parameters) trained on the entire coarse-grained training dataset, and tested on 10 different pure random noise. The model can gradually de-noise and generate images from pure noise(left) to clear images(right) over 10 de-noising steps.

batch, the model will train $10 \times 16 = 160$ times.

4.4 Test results

As Figure 7 and Figure 8 show, my final model (15.9 parameters) can generate clear images of size 96×96. For different noises, the model is able to generate different types of flowers. No colour shift in the generated images. As we can see in each de-noising step, the noise becomes less and less and the image becomes clearer and clearer, which demonstrates that the model has the ability to identify and remove noise.

4.5 Problems solved in my development process

I would like to share some issues I encountered during the process of the development of my UNet de-noising model, as well as how I solved them.

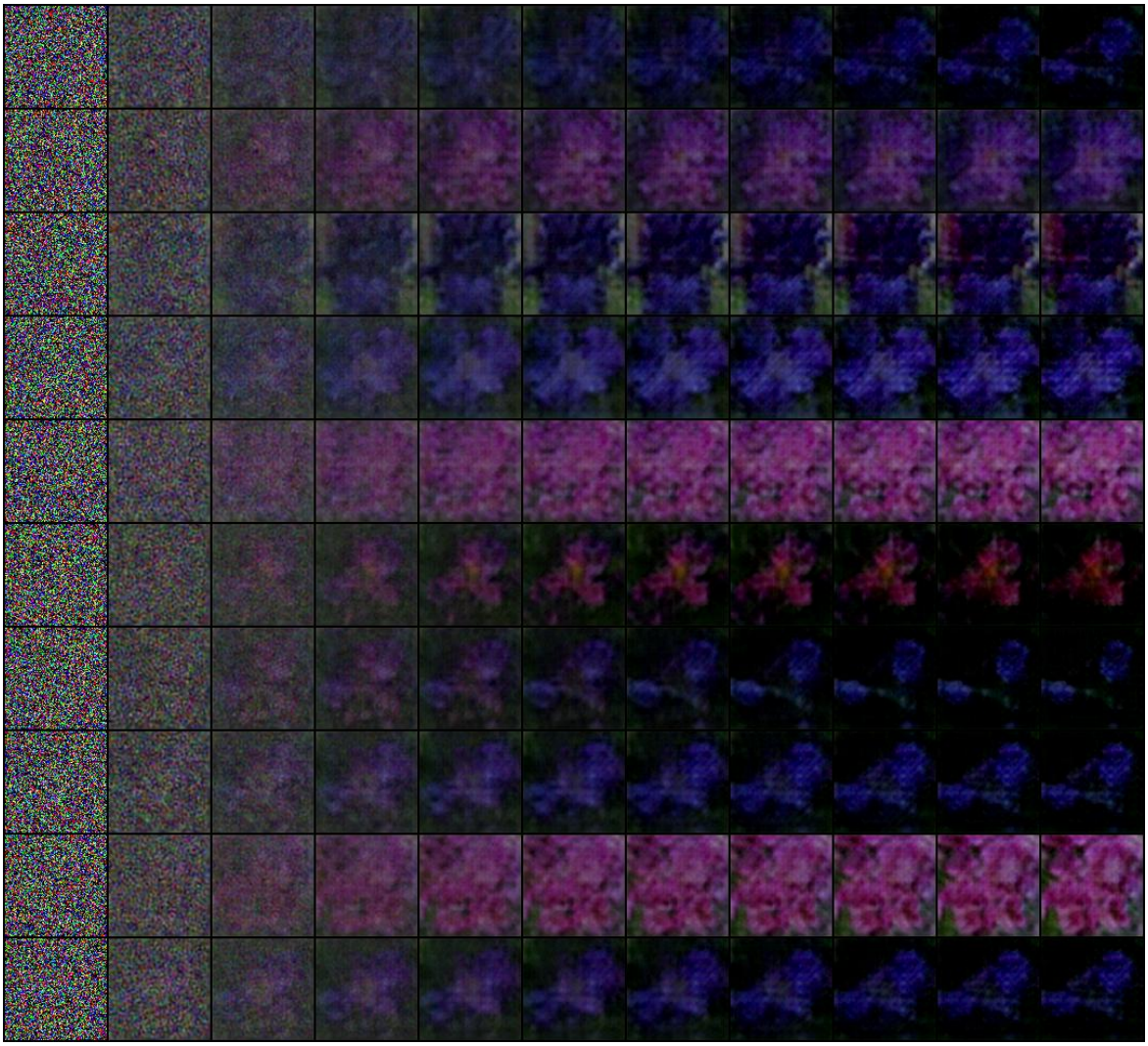

Figure 9: Problem: generated images getting darker and darker

The first problem I encountered was the color shift of the generated images. The generated images are getting darker and darker, as Figure 9 shows.

The reason for this problem is that when adding random noise to images, the color of pixels will become less than 0 or greater than 1. If I use this data for training, the model will learn to generate pixels that are less than 0 or greater than 1, which is not desired. The way to solve this problem is to clamp the colour of the noisy images between 0 and 1 before feeding it to the model.

Another problem is that once the model is trained and generates images, it prefers to generate red and purple flowers instead of other colours, as Figure 10 shows. I thought that the problem was that red or purple flowers dominate the dataset, but after examination, RGB of the pixels account for 40%, 34%, and 26%, respectively. Although red accounts for 40% but it should not make the model always generate red and purple flowers. Finally, I figured out the reason for this problem is that when preparing the random noise for the model, I just created random noise and clamped

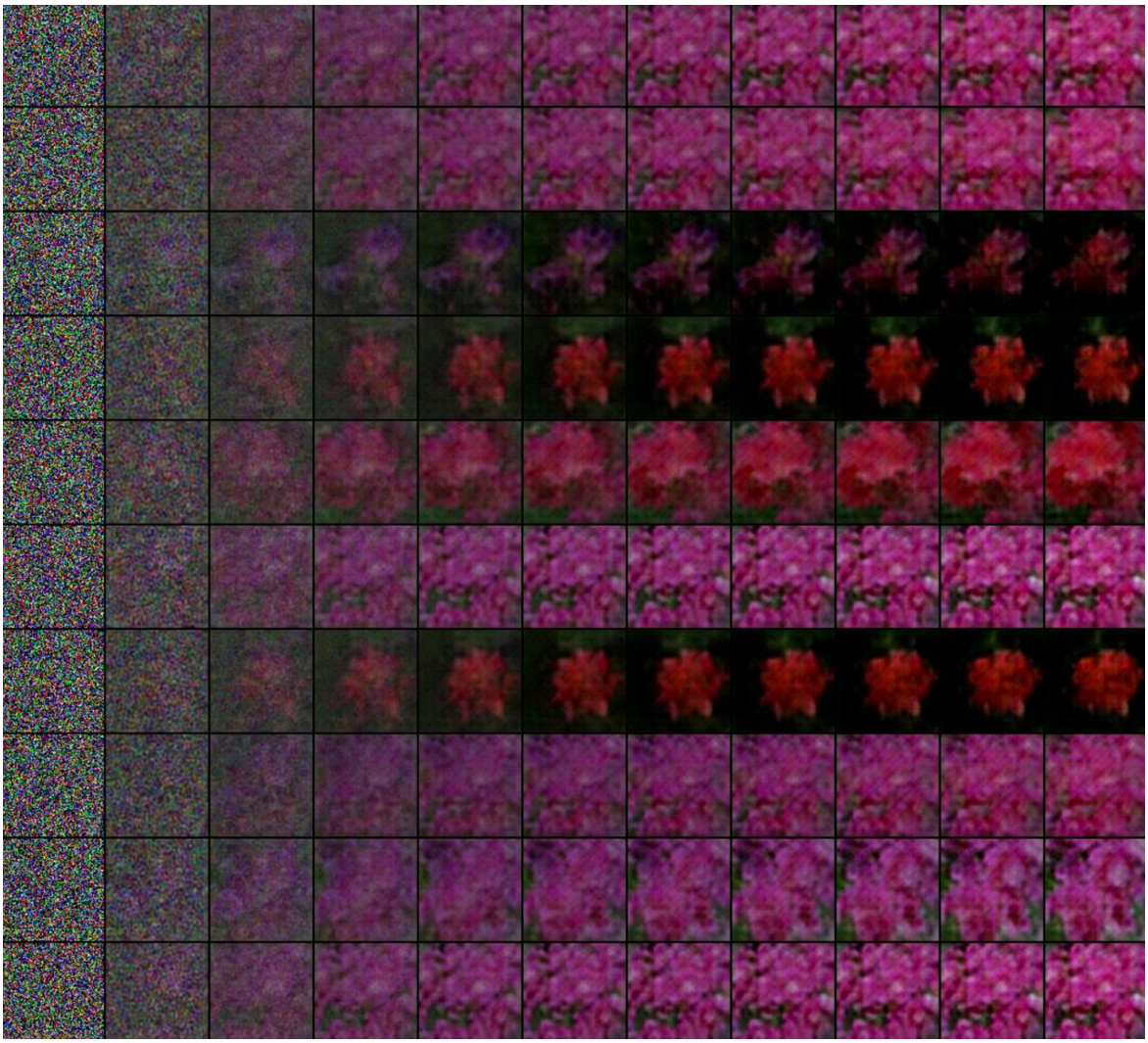

Figure 10: Problem: model always generates purple and red colour

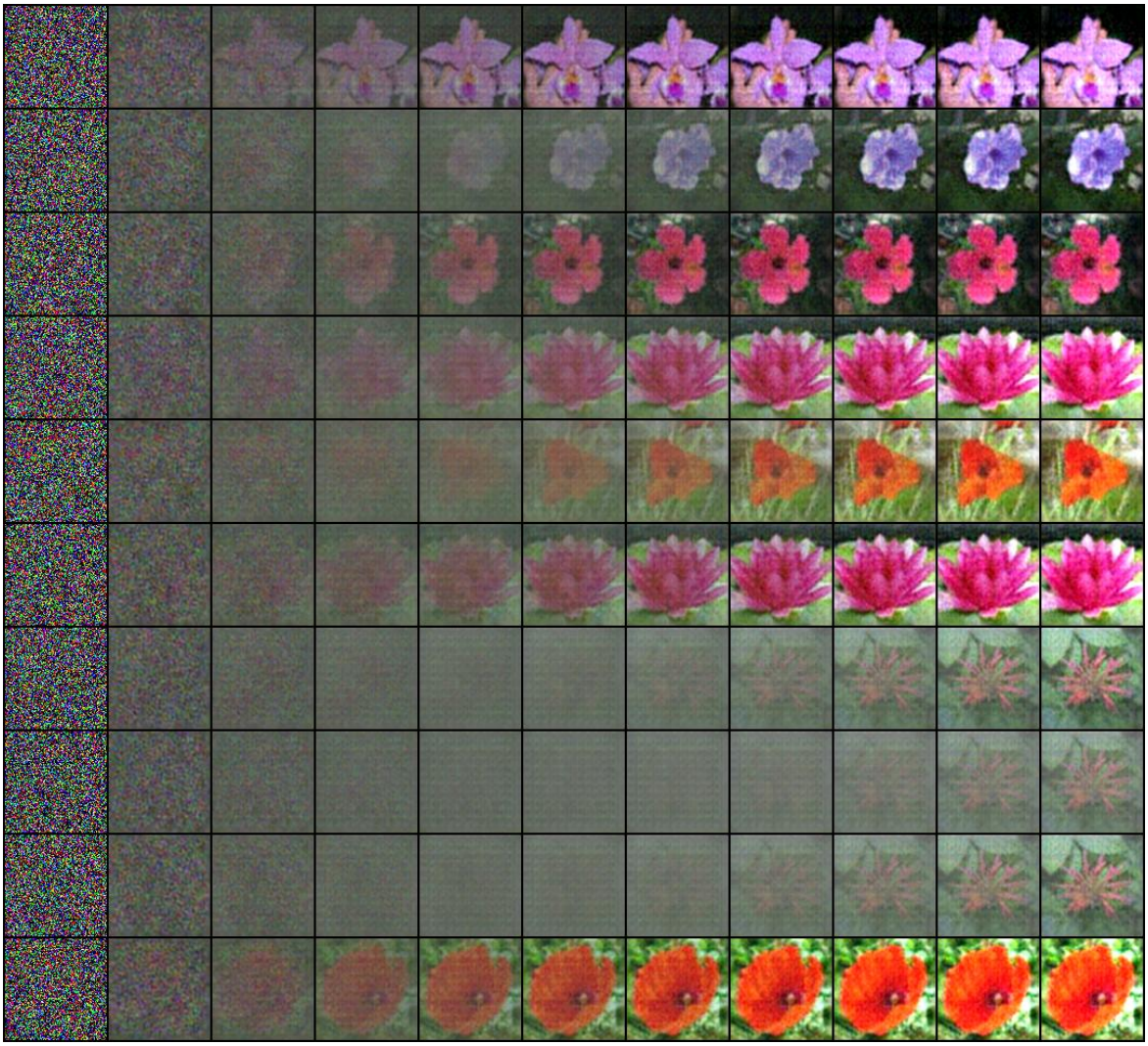

Figure 11: A simple model (5 million parameters) can generate very good images when training on only 50 image samples (even though a few of them are blurry)

it between 0 and 1. Because random noise created by *torch.randn()* has a mean of 0 by default, if I clamp it directly between 0 and 1, more value below 0 will be clamped while less value more than 1 is clamped. So the model will have colour shift. To fix the problem, when generating random noise, the mean of noise should be 0.5 instead of 0.

Another problem is that I can generate very clear images if I train the model with only 10 or 50 image samples, as Figure 11 shows, but if I train the same model on the entire dataset, the images generated are not clear and dark(Figure 12). The problem is that my previous model is too simple (5 million parameters) to capture enough features of larger image data and will get confused when training with lots of data. I made the UNet model more complex by adding more layers and more channels in the layers as my final model (15.9 million parameters). After making the model more complex, the image generated on the entire dataset becomes much clearer.

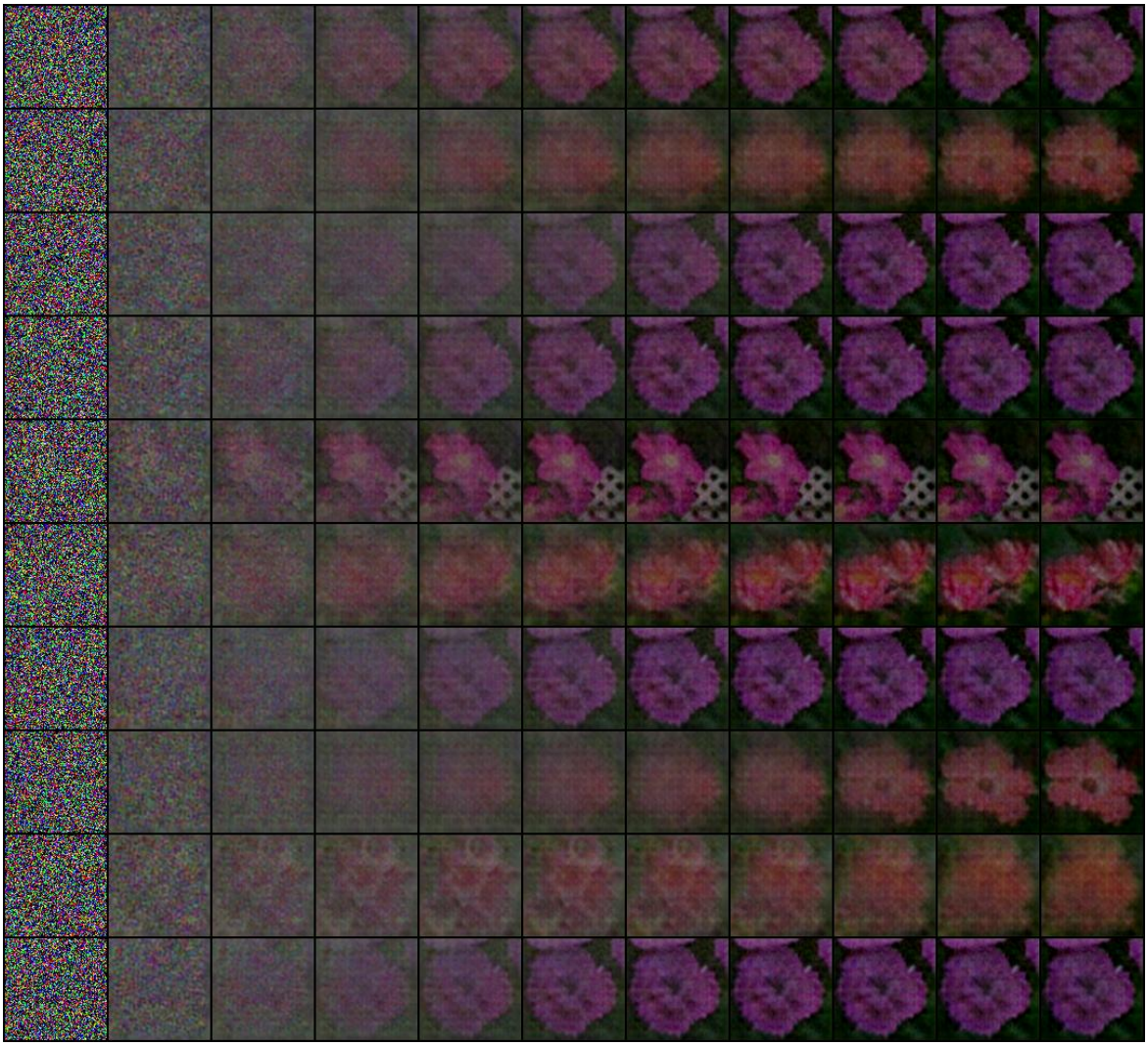

Figure 12: Problem: the simple model (5 million parameters) performs worse when training on the entire dataset compared to training on only 50 image samples

5 Conclusion

In this assignment, I developed a convolutional neural network(CNN) model for image classification, an auto-encoder to encode images to latent space and decode it back to the original image, as well as a UNet de-noising model that can generate clear images from pure random noise over multiple de-noising steps.

In finishing the assignment, I spent lots of effort on tuning the hyper-parameters and testing the results, identifying and fixing problems encountered during the development period.

The hardest part of developing deep neural network models is there are too many different combinations that can produce different results. It is too hard to think of a 'perfect' combination of architecture, data preparation, hyper-parameters, and training methods.

As a result, it is always a good idea to start with a simple model with a simple dataset. For example, when I was developing the UNet de-noising model, I started with training a very simple model(less than 1 million parameters) on only 1 flower sample, and see if it can reverse a pure noise to this flower image. If it can, I further train it to 10 samples and see if it can generate different images from different random noises. Once successful, I refined the model and trained it on 20 samples, 50 samples, 100 samples, and then on the entire dataset. Over time, the model architecture becomes more and more complex to complete the more and more complex tasks. By this means, I can not only learn how the model works from scratch, but also gradually build a model that is capable of completing complex tasks.