NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4042: Neural Networks and Deep Learning

Image Colourization

Chua Gim Aik (Cai JinYi) (U2022142B)
Tan Shu Hua, Samantha (U2021180J)
Venkataraman Sidhaarth (U2021808J)

1. Introduction & Problem Statement

For our Neural Networks group project, we chose something fun, interesting, challenging and most importantly within our capabilities as beginners. Our problem choice, **image colourization** will utilise several techniques that we have just learned in class like CNN, autoencoders and GAN, making this an exciting project to apply all the new techniques we've been taught!

Image colourization involves the transformation of a black and white image into a coloured image. While black-and-white photos are treasured possessions for many, they can appear dull and lifeless. Adding colour can make images more vibrant, realistic, and bring out the importance of the moment captured. Colourized images have many uses, from image recognition to object detection. Research on colorization methods covers various fields such as historical photograph restoration, animation design, and even infrared image processing.

Whilst turning a colour picture to black and white is simple, the reverse process is far more challenging. Adding colour without a reference depends on imagination, often resulting in unrealistic colours. It's a complex problem due to the varied colours an object might have, making the task intriguing.

Recently, deep learning has shown promise in image processing due to its excellent feature extraction capabilities. In 2015, the first deep learning-based image colorization method was introduced. Since then, these algorithms have outperformed traditional solutions and continue to advance. In this project, we explore 3 main different deep learning models, including a plain vanilla CNN, an Autoencoder, and a GAN model. Our objective is to find the best model for this task.

2. Dataset

We used a dataset of landscape images, consisting of streets, buildings, mountains, glaciers, trees etc and their corresponding grayscale image in two different folders. The dataset consists of 7128 images. You can download the dataset we used here: https://www.kaggle.com/datasets/theblackmamba31/landscape-image-colorization.

3. Methodology

We've learned various methods for image-learning tasks in class. Namely, CNN, autoencoders and GAN, why don't we start small by seeing how CNN performs? Spoiler alert: Since we will be covering multiple models, we will skip over some details for all but our final model for report brevity.

4. Convolutional Neural Network (CNN)

4.1. Why CNN?

Convolutional Neural Networks (CNNs) have emerged as powerful tools in the field of computer vision, demonstrating remarkable capabilities in tasks such as image recognition, object detection, and image generation. CNNs are a class of deep neural networks designed to process and analyse visual data, inspired by the structure and functioning of the human visual system. We aim to leverage on its convolutional layers to learn features and spatial dependencies from grayscale images, enabling it to predict accurate colour information. The hierarchical representation of features and the end-to-end learning process make CNNs a natural choice for image colourization, offering a data-driven and efficient solution to this visually compelling task.

4.2. Architecture

Our CNN model is structured as a sequential model, composed of convolutional layers, max-pooling layers, and fully connected layers. The model takes grayscale images of size 256 x 256 as input, and

outputs colourized images with the same dimensions. The convolutional layers are essential for extracting features from the input images. Max-pooling layers are incorporated to downsample the spatial dimensions of the feature maps, reducing computational complexity and preserving essential features. Max-pooling is applied after each convolutional layer. The intricacies of the model can be found in Appendix A.

The model is compiled using the Mean Squared Error (MSE) loss function, which measures the difference between the predicted and actual pixel values. The Adam optimizer with a learning rate of 0.001 is employed to minimise the loss. The accuracy metric is used for monitoring the training process.The model is trained using a dataset of size 700 consisting of grayscale images and their corresponding colourized counterparts. Training is performed for 100 epochs with a batch size of 16. Early stopping is implemented with a patience of 3 epochs to prevent overfitting, and the model weights are restored to the best-performing configuration.

4.3. Results

Despite its efficacy in learning complex relationships between grayscale and colour images, our model did not perform very well. One significant drawback pertains to the resultant image quality, notably the low resolution of the predicted colourized images as seen in Figure 1 below. The model's output exhibited a reduction in image quality, with blurriness and a lack of fine details.

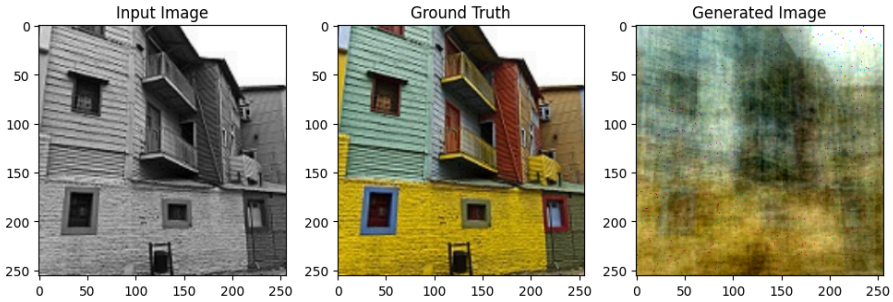


Figure 1: Prediction from CNN model

This decline in resolution is likely attributed to the downsampling effects introduced by the convolutional and pooling layers, which result in a loss of spatial information. The downsampling process can hinder the network's ability to capture and retain intricate details crucial for high-resolution colorization. Additionally, the limitations imposed by the network's architecture and the fixed-size convolutional filters may constrain the model's ability to preserve fine-grained details in the colorization process, resulting in images that lack the sharpness and clarity found in higher-resolution inputs.

Thus, we decided to introduce upsampling techniques and skip connections in our subsequent model, in hopes of helping to recover the spatial information lost during downsampling and retain the finer details.

5. Autoencoder

5.1. Why Autoencoder?

Autoencoders represent a distinctive category of deep learning architectures comprising two primary components: an encoder and a decoder. The encoder employs a sequence of CNNs and downsampling techniques to learn a compressed, lower-dimensional representation of the input data. Conversely, the decoder, utilising CNNs and upsampling methodologies, endeavours to reconstruct the data from these learned representations. A proficiently trained decoder possesses the capability to regenerate data that closely resembles or matches the original input data. Autoencoders are commonly employed for diverse tasks such as anomaly detection, denoising images, and colourizing images - perfect for our task at hand.

5.2. Architecture

Encoder: The encoder is responsible for compressing the input image into a lower-dimensional latent space, and consists of convolutional layers, each followed by a ReLU activation function for non-linearity.

Decoder: The decoder layer reconstructs the original image from the compressed representation, and consists of transposed convolutional layers which increase the spatial dimensions while reducing depth.

Skip Connections: Some layers in the network were skipped, feeding the output of one layer as the input to the next layers. This provides a way to bring features from the encoder directly to the decoder. This was done to mitigate the problem of vanishing gradients by allowing an alternate shortcut path for the gradient to flow through. It allows the model to preserve spatial features that may be lost during the encoding process. Dropout layers were also applied after the skip connections were concatenated to the feature maps to ensure the model does not become overly reliant on any one path. Refer to Appendix B for more exact details on our Autoencoder model architecture.

5.3. Results

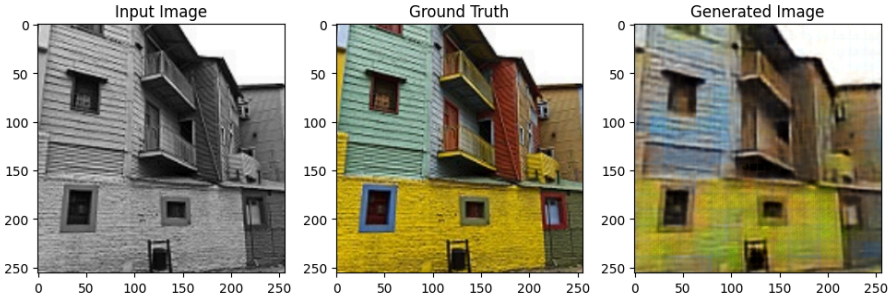


Figure 2: Prediction from Autoencoder model

From the results, the Autoencoder model seems to have a higher colour accuracy than the plain CNN and preserves details better, likely due to the encoding process capturing essential information that more closely resembles the ground truth. It also maintains the structural integrity of the image better, which could be due to the skip connections that help in retaining spatial information lost during encoding.

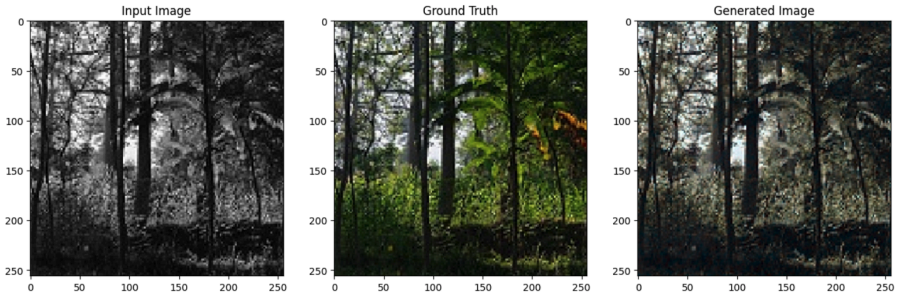


Figure 3: Prediction from Autoencoder model

However, we found that there were several instances such as above where the Autoencoder model produced significantly muted colours, with the greens and browns not as vivid as the ground truth. This model still seems to struggle with fine details and colour bleeding. The generated image also shows signs of noise and non-uniform colour patches throughout the image. The model did not display sufficient spatial coherence with differentiation between foreground and background elements, resulting in a somewhat distorted colourisation. This could be due to a limited learning capacity and lack of global context (autoencoders often learn localised features well but may miss out on the global context of the image which is essential for learning how different parts of the image relate to each other in terms of colour and texture). Given these issues, we felt it might be worthwhile to explore the GAN model next.

6. Generative Adversarial Network (GAN)

6.1. Why GAN?

GAN is a type of neural network typically used for generative modelling, consisting of 2 main parts: the generator and the discriminator. These 2 networks are trained simultaneously in a competitive setting.

The generator network generates new data instances, trying to mimic the distribution of real data. It learns to create data that is indistinguishable from actual data. The discriminator network evaluates the data for authenticity, trying to distinguish real data from data created by the generator.

The training is kind of like a tug of war, where the generator learns to produce more and more realistic data while the discriminator gets better at telling real data from fake. This process continues until the discriminator can no longer easily differentiate real data from the fakes, indicating that the generator is producing high-quality data.

This will be our final model and we also found it to be our best, so let's go through it with a little more depth.

6.2. Methodology & Data Processing

We skipped over these for our last two models, but we will emphasise it here to avoid repeating ourselves and while the approach may differ slightly, the fundamental idea remains the same.

6.2.1. RGB vs L + ab Formulation

The first thing to consider is how exactly will the model predict the colours of Black and White images? Our initial instinct was to consider RGB which refers to the 3 main colour channels of most computer images: Red, Green and Blue. We found strong support for RGB online for many image tasks but we eventually learned that specifically for image colourisation, it might be better to consider an alternative colour encoding scheme: LAB colours. Now, there are many colour schemes, but let's just explore LAB.

LAB stands for Luminance (lightness information) + AB (which are green-red and yellow-blue colour intensity channels). This makes a big difference for our task at hand. If we use RGB, our model needs to predict 3 channels (R,G and B) per pixel, and must understand and reconstruct the entire colour information from scratch. Number of predictions exponentially increases. If we use L+ab, our model's task focuses on adding colour by predicting 2 values, a and b per pixel, making the combination of possible predictions significantly lower. This will help reduce the number of training parameters and potentially the model performance as well since it's a lot easier for the model as it basically gets to guess fewer numbers to achieve the same objective.

So, for our problem, we take coloured images, extract the L channel, and use it as input while keeping the a + b channels as "labels" to make our problem semi-supervised (i.e. generator & discriminator model will see the true image). We will get to see some differences between GAN trained using RGB vs LAB encoding.

6.2.2. Data Normalisation

According to GAN hacks suggested by experts, normalising our images is recommended to help stabilise training. Even though our input value range already exists on a fixed scale (0 to 255 for each pixel), we decided to normalise our input tensors for each image channel to a range of -1 and 1. We were mindful to denormalise our images when trying to obtain the original image for presentation afterwards. We found that data normalisation did improve our model training slightly.

6.2.3. Adding Noise

Adding noise to images is a very common practice to improve a model's robustness and can act as a form of regularisation. Since we have very limited input (7k+ images), we introduced noise to our data to help prevent overfitting. We did this to our images mainly in two ways: firstly, by adding random uniform noise to our images (which produces a static-like texture in our images) and secondly, by randomly flipping our images horizontally. Note: there are many image augmentation techniques, but these are the main ones we used. We also found that adding noise to our images helped confuse our generator and discriminator models more during training and improved performance slightly.

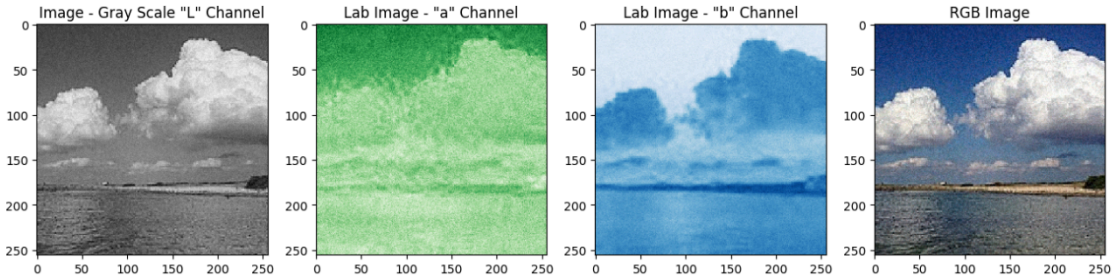


Figure 4: L + ab channel splitting and noise addition

6.2.4. Train-test Split

We did a very simple 70-30 train-test split, the generator and discriminator will train on 70% of the data and we will validate it against the remaining 30% (for early stopping, see Section 6.3.8.).

6.3. Training Strategy

This part is quite complex (for us), we took inspiration from what others have tried and added our own twists.

6.3.1. Convolution

In order to help our generator and discriminator extract features like any CNN, we need to perform convolution by downsampling. In the case of our generator, we also need to perform upsampling to build upon learned features and generate features to restore colours to grayscale images.
We used 2D convolution and Convolution Transpose layers for upsampling and downsampling respectively.
Our convolution layers both use striding of 2 and kernel filter size of 4 by 4 with padding of 1. These were the recommended settings we found from reviewing other people's work that we found that work for us.

6.3.2. Batch Normalisation

Normalising the outputs of each layer can improve regularisation and model training. We used Pytorch's BatchNorm2d layer and applied this layer for most of our downsampling and upsampling layers.

6.3.3. Dropout Layers

To better train our generator, we also added more noise in the form of dropout layers, we applied dropout layers with a dropout probability of 0.5 to the first three downsampling and upsampling layers. This will help our generator avoid relying too much on certain features and learn more variety of features.

6.3.4. Activation Functions

For our activation functions, experts recommended using LeakyRelu for both the Generator and Discriminator. We used LeakyRelu for all our upsampling and downsampling layer activation except the final layer of our Generator where we use tanh().

6.3.5. Skip Connections

This idea is mainly inspired by the U-net architecture. We added skip connections to better preserve the flow of our gradients and this will aid in recovering fine-grained details. Adding skip connections helped our generator produce less blurry images.

6.3.6. Batch Size

We experimented with this a lot, and decided to train our model on mini-batch sizes of 8-16. Not only does increasing batch size further cause computation problems such as GPU out-of-memory issues, but it also hurts model training.

6.3.7. Learning Rate

We initially did the standard 0.001 but after experimentation and affirmation from online opinions, we found 0.0002 to be more reliable for stable training.

6.3.8. Early Stopping

It was extremely difficult to perform early stopping, our first instinct was to use GAN loss like we are used to in traditional deep learning methods. But after some research, we found that GAN loss is unreliable for evaluating GAN models. Instead, experts recommend other metrics such as Frechet Inception Distance (FID) for early stopping.

FID calculates the distance between real and generated images by looking at both images' distribution. We will use this as our early stopping metric instead.

However, this is a very expensive operation as the FID is also a pre-trained model that requires forward computation of image tensors. To save on computation, we will only apply FID for every set number of epochs and call off training if FID doesn't improve for a while.

A lower FID score is desired for our generator model as it means it is producing images closer to the original.

6.4. Model Architecture Overview

Generator: The generator takes a grayscale image as an input (L channel). It uses a series of downsample layers to progressively decrease the spatial dimensions while increasing the depth of the feature maps. This series includes convolutional layers with increasing channel sizes and applies dropout to introduce noise which helps with generating more diverse outputs. The smallest spatial dimension is reached at the bottleneck, where depth is maximum. Following which, a series of upsample layers progressively increase the spatial dimensions through transposed convolutions. Like in the Autoencoder, skip connections that preserve spatial information have been included as well. The last transposed convolutional layer outputs 2 channels representing the a and b channels, which are then passed through a Tanh activation function to produce the final colourisation result.

Discriminator: The discriminator receives both the grayscale input image (L) and the colour channels (a and b), which could be either the ground truth or the output from the generator. It processes the concatenated input through several downsample layers, similarly decreasing spatial dimensions while increasing the feature map depth. Additional zero-padding layers and a convolutional layer further process the features. The last convolutional layer outputs a single channel feature map, which is intended to provide a real/fake score for the input image, indicating whether the colorization is real or generated.

Both the generator and discriminator use LeakyReLU activations and batch normalisations except in the first layer to stabilise training. Refer to Appendix C & D for exact specifics on the generator and discriminator model architecture.

6.4.1. Loss Function

Generator loss: Gan Loss: Uses 'BCEWithLogitsLoss, which combines a sigmoid layer with the binary cross entropy loss, calculated between the discriminator's output on the generator's images and an array of ones, which represent the target label for real images. The goal here is to make the discriminator classify the generated images as real (one).
L1 Loss: The mean absolute error between the generated image and the target image. This loss ensures that the generated images are not just classified as real by the discriminator but are also close to the real images.

Total generator loss: The sum of the GAN loss and the weighted L1 loss (scaled by LAMDA a hyperparameter to balance the two loss components). The L1 loss has a regularising effect, encouraging the generator to produce images that are structurally similar to the target images.

Discriminator loss:
The discriminator loss has 2 components, real loss (using BCEWithLogitsLoss) and generated loss as well, summed together.

The balance between these adversarial components is what allows GANs to generate high quality outputs, so we played around with the LAMBDA parameter to ensure that the generator's training parameter did not solely focus on fooling the discriminator but also produced high quality images.

6.5. Results & Observations

We observed decent results with this model, and learned a few things…

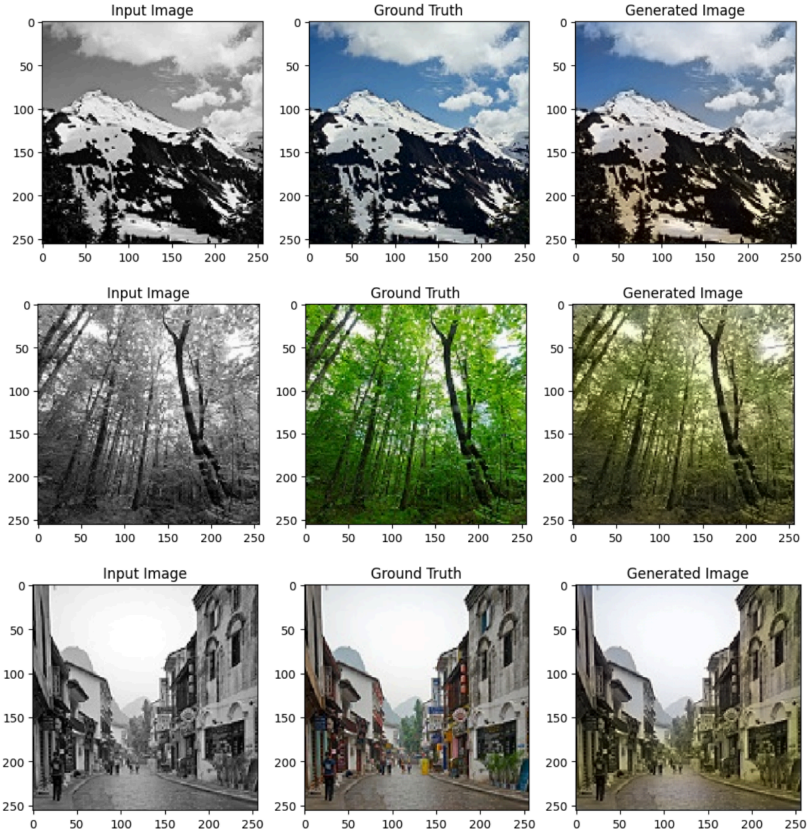


Figure 5, 6 & 7: L + ab channel splitting and noise addition

General observations:

Our GAN model performed significantly better than the previous models we have tried, it produced less blurry images with richer colour diversity. That being the case, it's still not perfect as we found that our models have a bias towards blue & white images, this could be due to our data quality but in general, our model was able to churn out coloured images of skies, sea, mountains, snow, etc. really well as those elements contain a lot of white and blue elements. Our model performed moderately for forests, trees and greenery as it often refused to take risks to choose rich green colours and opted for lighter greens. Our models performed poorly for urban architectures and images that people, this is almost certainly due to our data quality since we train mostly on landscape and nature images, but another issue arises from the fact that human settlements and structures often have rare bright colours like yellow and red that don't occur as naturally in nature.

6.5.1. RGB vs Lab

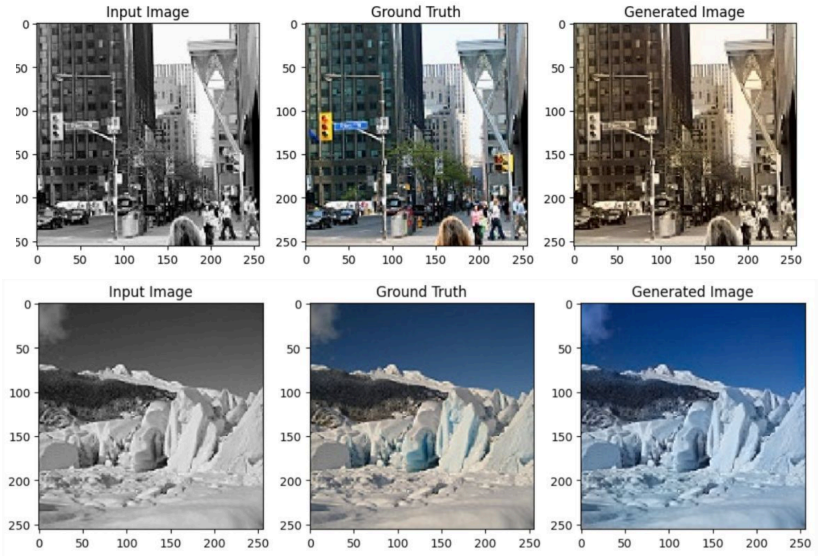


Figure 8 & 9: RGB GAN Model

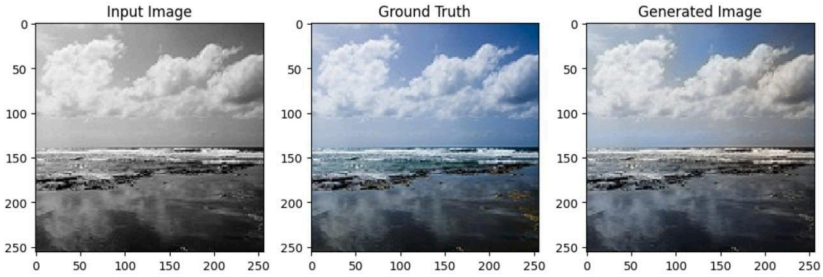


Figure 10: LAB GAN Model

Transitioning from an RGB to Lab colour space for training the model had a transformative effect on the results. The Lab colour space allowed for more nuanced colorization that more closely mirrored human perception as compared to the GAN run in the RGB colour space as can be seen above. The generated images exhibited a richer and more accurate colour profile. We found that RGB had heavy biases to one of the colour channels (mostly green for us). This is an excellent example of how proper problem formulation can improve model training. Takeaway: don't just look at models and parameters, also think about how we can reduce or make a problem simpler.

6.5.2. Overfitting vs Early Stopping

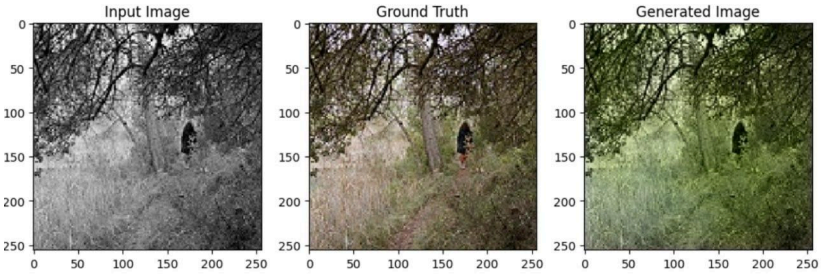


Figure 11: GAN Model without Early Stopping (Overfitted)

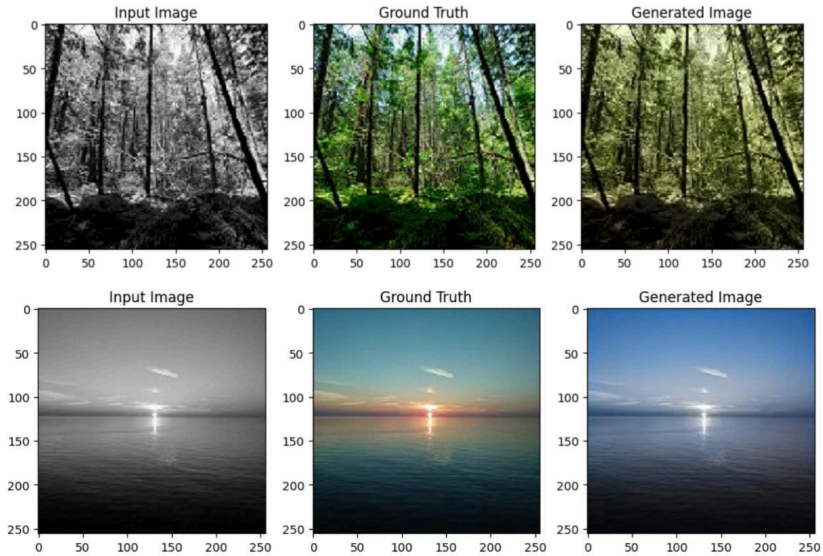


Figure 12 & 13: GAN Model with Early Stopping

Initially we found that the generated image captured the general structure of the scene, but exhibited a lack of colour diversity and realism, with the colours appearing muted and blending into each other. This suggested that the model has memorised specific patterns from training data which did not generalise well to unseen images. In contrast, the model with early stopping *(Figure 8 & 9)* showed a marked improvement. The generated image, while still easily differentiable from the ground truth, had more vibrant and distinct colours, demonstrating the efficacy of early stopping as a regularisation technique.

6.5.3. Effects of adding noise

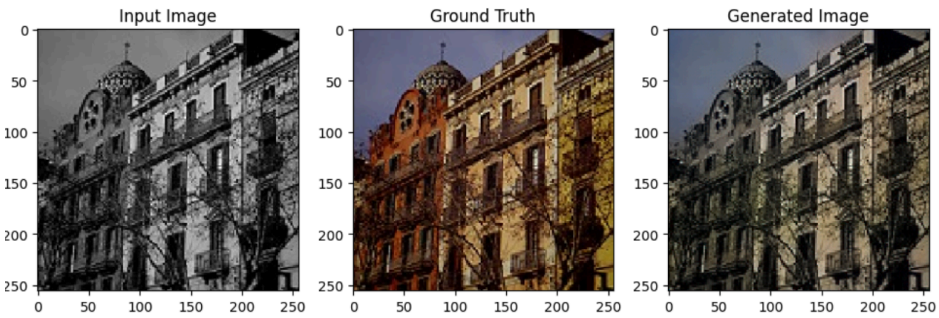


Figure 14 : GAN Model (No Noise)

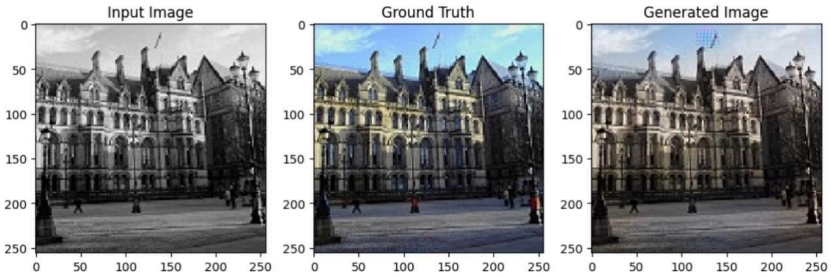


Figure 15: GAN Model (With Noise)

Introducing noise to the training process, aimed at enhancing robustness of the model, yielded mixed results. While it prevented the model from learning a noise free, perfect representation of the data which could contribute to overfitting, it also introduced some degree of uncertainty to the process. This trade-off resulted in a model that was more resilient to variations in input data, but at the cost of some colour fidelity. Adding noise is thus expected to make the model more robust, but at the cost of reduced colorization sharpness and accuracy.

7. Limitations and Future Improvements

7.1. Conservative model problem

Across all our attempts to tune our GAN model, we found that if it didn't suffer from overfitting, it suffered from being too conservative. The conservative model problem occurs when the model predicts very neutral colours like brown or grey. This fear of taking risks might be linked to our use of L1 loss for our Generator model. When it's unsure of what colour to use, the generator will choose conservative estimates by choosing averages, in colour talk, this could be brown, grey, etc.

To overcome this, we should explore other loss functions and consider tuning our LAMBDA parameter further to reduce the emphasis on L1 loss.

7.2. Model's colour bias problem

Our model is still biased to certain colours while underexposed to certain colours even after several rounds of tuning, we suspect this is due to our image quality. To fix this, we should consider our image quality and diversity. Upon inspection, we found several coloured images that exist in black and white. Adding more images of urban structures and people will certainly help as well.

7.3. Combination with other models

One thing we did not try was combining image colourisation with other model techniques to improve our overall solution. For instance, we could combine our model with pre-trained image refinement models to get better image quality or even perform transfer learning with image segmentation models to help our model better learn colourisation features for our problem.

8. Conclusion

In summary, we have tried various model architectures, namely: CNN, Autoencoder and GAN architectures, and found that our GAN model achieved the best results. We have also applied various deep learning techniques (such as early stopping, normalising, noising our images, etc.) to improve our models and found relative success in doing so. Our learning journey has also documented that we have iteratively built upon our solution to improve our project and achieved decent results along the way although it is clear that there is still room for improvement. Thank you for reading our report and reviewing our project! :)

9. References

Shariatniam, M. (2020, Nov 19). *Colorizing black & white images with U-Net and conditional GAN — A Tutorial,* https://towardsdatascience.com/colorizing-black-white-images-with-u-net-and-conditional-gan-a-tutorial-81b2df111cd8

Nelson, J. (2020, Mar 19). *Why add noise to images for machine learning?* https://blog.roboflow.com/why-to-add-noise-to-images-for-machine-learning/

Mittal, T. (2019, Aug 24). *Tips On Training Your GANs Faster and Achieve Better Results.* https://medium.com/intel-student-ambassadors/tips-on-training-your-gans-faster-and-achieve-better-results-9200354acaa5

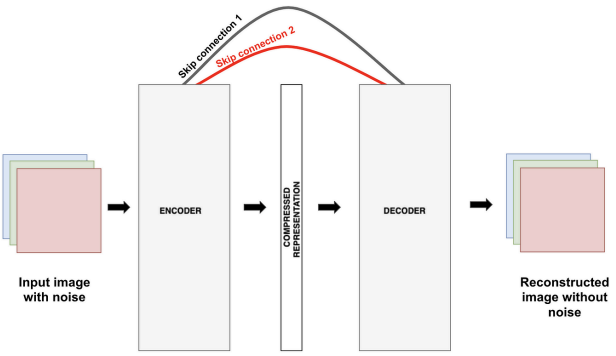
Chintala, S., Denton, E., Arjovsky, M., Mathieu, M. (2020, Mar 5). *How to Train a GAN? Tips and tricks to make GANs work.* https://github.com/soumith/ganhacks

10. Appendix

Appendix A: CNN Model Architecture

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Conv_01 (Conv2D)            (None, 256, 256, 64)      1792

 max_pooling2d (MaxPooling2  (None, 128, 128, 64)      0
 D)

 Conv_02 (Conv2D)            (None, 128, 128, 128)     73856

 max_pooling2d_1 (MaxPoolin  (None, 64, 64, 128)       0
 g2D)

 Conv_03 (Conv2D)            (None, 64, 64, 256)       295168

 max_pooling2d_2 (MaxPoolin  (None, 32, 32, 256)       0
 g2D)

 Conv_04 (Conv2D)            (None, 32, 32, 512)       1180160

 max_pooling2d_3 (MaxPoolin  (None, 16, 16, 512)       0
 g2D)

 flatten (Flatten)           (None, 131072)            0

 dense (Dense)               (None, 512)               67109376

 dense_1 (Dense)             (None, 196608)            100859904

 reshape (Reshape)           (None, 256, 256, 3)       0

=================================================================
Total params: 169520256 (646.67 MB)
Trainable params: 169520256 (646.67 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

Appendix B: Autoencoder Model Architecture



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 160, 160]	640
Conv2d-2	[-1, 64, 80, 80]	36,928
Conv2d-3	[-1, 128, 40, 40]	73,856
Conv2d-4	[-1, 256, 20, 20]	295,168
ConvTranspose2d-5	[-1, 128, 40, 40]	295,040
Dropout-6	[-1, 256, 40, 40]	0
ConvTranspose2d-7	[-1, 64, 80, 80]	147,520
Dropout-8	[-1, 128, 80, 80]	0
ConvTranspose2d-9	[-1, 128, 160, 160]	147,584
Dropout-10	[-1, 192, 160, 160]	0
ConvTranspose2d-11	[-1, 15, 160, 160]	25,935
Conv2d-12	[-1, 2, 160, 160]	290

Appendix C: GAN Generator Model Architecture

Layer (type)	Output Shape	Param #
Conv2d-1	[16, 64, 128, 128]	1,024
Dropout-2	[16, 64, 128, 128]	0
LeakyReLU-3	[16, 64, 128, 128]	0
downsample-4	[16, 64, 128, 128]	0
Conv2d-5	[16, 128, 64, 64]	131,072
BatchNorm2d-6	[16, 128, 64, 64]	256
Dropout-7	[16, 128, 64, 64]	0
LeakyReLU-8	[16, 128, 64, 64]	0
downsample-9	[16, 128, 64, 64]	0
Conv2d-10	[16, 256, 32, 32]	524,288
BatchNorm2d-11	[16, 256, 32, 32]	512
Dropout-12	[16, 256, 32, 32]	0
LeakyReLU-13	[16, 256, 32, 32]	0
downsample-14	[16, 256, 32, 32]	0
Conv2d-15	[16, 512, 16, 16]	2,097,152
BatchNorm2d-16	[16, 512, 16, 16]	1,024
LeakyReLU-17	[16, 512, 16, 16]	0
downsample-18	[16, 512, 16, 16]	0
Conv2d-19	[16, 512, 8, 8]	4,194,304
BatchNorm2d-20	[16, 512, 8, 8]	1,024
LeakyReLU-21	[16, 512, 8, 8]	0
downsample-22	[16, 512, 8, 8]	0
Conv2d-23	[16, 512, 4, 4]	4,194,304
BatchNorm2d-24	[16, 512, 4, 4]	1,024
LeakyReLU-25	[16, 512, 4, 4]	0
downsample-26	[16, 512, 4, 4]	0
Conv2d-27	[16, 512, 2, 2]	4,194,304
BatchNorm2d-28	[16, 512, 2, 2]	1,024
LeakyReLU-29	[16, 512, 2, 2]	0
downsample-30	[16, 512, 2, 2]	0
Conv2d-31	[16, 512, 1, 1]	4,194,304
BatchNorm2d-32	[16, 512, 1, 1]	1,024
LeakyReLU-33	[16, 512, 1, 1]	0
downsample-34	[16, 512, 1, 1]	0
ConvTranspose2d-35	[16, 512, 2, 2]	4,194,304
BatchNorm2d-36	[16, 512, 2, 2]	1,024
Dropout-37	[16, 512, 2, 2]	0
LeakyReLU-38	[16, 512, 2, 2]	0
upsample-39	[16, 512, 2, 2]	0
ConvTranspose2d-40	[16, 512, 4, 4]	8,388,608
BatchNorm2d-41	[16, 512, 4, 4]	1,024
Dropout-42	[16, 512, 4, 4]	0
LeakyReLU-43	[16, 512, 4, 4]	0
upsample-44	[16, 512, 4, 4]	0
ConvTranspose2d-45	[16, 512, 8, 8]	8,388,608
BatchNorm2d-46	[16, 512, 8, 8]	1,024
Dropout-47	[16, 512, 8, 8]	0
LeakyReLU-48	[16, 512, 8, 8]	0
upsample-49	[16, 512, 8, 8]	0
ConvTranspose2d-50	[16, 512, 16, 16]	8,388,608
BatchNorm2d-51	[16, 512, 16, 16]	1,024
LeakyReLU-52	[16, 512, 16, 16]	0
upsample-53	[16, 512, 16, 16]	0
ConvTranspose2d-54	[16, 256, 32, 32]	4,194,304
BatchNorm2d-55	[16, 256, 32, 32]	512
LeakyReLU-56	[16, 256, 32, 32]	0
upsample-57	[16, 256, 32, 32]	0
ConvTranspose2d-58	[16, 128, 64, 64]	1,048,576
BatchNorm2d-59	[16, 128, 64, 64]	256
LeakyReLU-60	[16, 128, 64, 64]	0
upsample-61	[16, 128, 64, 64]	0
ConvTranspose2d-62	[16, 64, 128, 128]	262,144
BatchNorm2d-63	[16, 64, 128, 128]	128
LeakyReLU-64	[16, 64, 128, 128]	0
upsample-65	[16, 64, 128, 128]	0
ConvTranspose2d-66	[16, 2, 256, 256]	4,098
Tanh-67	[16, 2, 256, 256]	0

Appendix D: GAN Discriminator Architecture

Layer (type)	Output Shape	Param #
Conv2d-1	[16, 64, 128, 128]	3,072
LeakyReLU-2	[16, 64, 128, 128]	0
downsample-3	[16, 64, 128, 128]	0
Conv2d-4	[16, 128, 64, 64]	131,072
BatchNorm2d-5	[16, 128, 64, 64]	256
LeakyReLU-6	[16, 128, 64, 64]	0
downsample-7	[16, 128, 64, 64]	0
Conv2d-8	[16, 256, 32, 32]	524,288
BatchNorm2d-9	[16, 256, 32, 32]	512
LeakyReLU-10	[16, 256, 32, 32]	0
downsample-11	[16, 256, 32, 32]	0
ZeroPad2d-12	[16, 256, 34, 34]	0
Conv2d-13	[16, 512, 31, 31]	2,097,152
BatchNorm2d-14	[16, 512, 31, 31]	1,024
LeakyReLU-15	[16, 512, 31, 31]	0
ZeroPad2d-16	[16, 512, 33, 33]	0
Conv2d-17	[16, 1, 30, 30]	8,193