Machine Learning Project
"Colorization"

Computer Science, Second Cycle
HT23-VT24, DT723A
Örebro University

Rickard Hallerbäck
rickard.hallerback@gmail.com

February 2024

Contents

1 Background **3**
- 1.1 Problem description . . . 3
- 1.2 Narrowing down the problem . . . 3
- 1.3 About the problem . . . 3
- 1.4 About the report . . . 3
- 1.5 About the author . . . 4

2 Dataset **4**
- 2.1 Rational behind dataset . . . 4

3 Data processing **5**
- 3.1 Color space . . . 5
  - 3.1.1 Normalization . . . 5
- 3.2 Code for loading image data . . . 5
  - 3.2.1 Image loader in Python . . . 5
- 3.3 Image size . . . 9

4 Modelling **9**
- 4.1 Network structure . . . 9
  - 4.1.1 Generator Network . . . 9
  - 4.1.2 Discriminator Network . . . 11
  - 4.1.3 Generative Adversarial Network . . . 13
  - 4.1.4 Reconstruction and adversarial loss . . . 13
- 4.2 Training in an adversarial manner . . . 14
  - 4.2.1 Model evaluation . . . 18

5 Results **19**
- 5.1 Adversarial effect . . . 19
- 5.2 Model complexity . . . 23
- 5.3 Model selection . . . 34

6 Lessons learned **36**
- 6.1 Importance of input processing . . . 37
- 6.2 Adversarial training . . . 37
- 6.3 Hyperparameter exploration . . . 37
- 6.4 Hardware evaluation . . . 38

1 Background

As part of the course in Machine Learning, DT723A in Örebro University, we have been encouraged to learn by doing and commit to a project. We were granted the opportunity to choose our own project idea. I chose to build an image colorizer program with deep learning. This report covers my journey on implementing an image colorizer program.

1.1 Problem description

Build a software program that can take a gray scaled image, and output a convincingly colorized version of that image. In this context, convincingly means that a human observer thinks the colorization looks realistic.

1.2 Narrowing down the problem

My computer is not the strongest but I will use it for training. I think that if I use a dataset with a specific type of images that my program can colorize, the model might not have to be so big. A more specialized model in my program can afford to be smaller, and run faster on my machine. I will focus on landscape pictures, and will be happy if the program manages to colorize such images.

1.3 About the problem

In my opinion, colorizing an image is no trivial task. The program needs to be based on an algorithm that is able to recognize that the sky is blue and that grass is green and doing this by writing a conventional computer program based on some kind of explicit logic is typically not known to work. Instead, one can use the machine learning technique called deep learning. In this way, the program is built by carefully curve-fit a non-linear mathematical function with enough parameters to cover the distribution of the dataset which hopefully also performs well in a general manner on data taken from a source not present in the dataset used for curve-fitting. The process of curve-fitting is in this sense called learning.

To me, this represents an exciting opportunity to dig deeper into the field of machine learning and deep learning in particular and build something I have never built before.

1.4 About the report

I want this report to be as comprehensible as possible, but I will assume that the reader is familiar with deep learning and is no stranger to deep learning concepts such as activation functions and different type of layers for example convolution layer or drop out layer. The theory of deep learning will mostly be out of scope of this report, and instead this will be more of a field report on how I solved this problem. I will share my thoughts on the process of desigining this system that solves the given problem and I will also share code snippets and results gathered along the way, such as tinkering with hyperparameters. The full code for the project is made available via this link: https://github.com/Ricardicus/image-colorizer, but I will make sure to share some key parts of the code in this report as well.

1.5 About the author

My experience in machine learning goes back to the year of 2015, where I underwent courses in the field at the university of Lund during my education as I pursued a Masters of engineering degree in electrical engineering. Back then, the field looked very promising and interesting. The general sense was that this was an evolving field that had huge potential. But this was before the recent ground breaking achievements were made such as ChatGPT. My masters thesis was about predicting the 90 day mortality rate of patients undergoing surgery for Esophagus Cancer, Esophagectomy, using deep learning in collaboration with surgeons at Lund University Hospital. To me, it seems like machine learning has matured a lot since then, since last I was up to date. When I did deep learning, for example, I implemented the backpropagation algorithm myself in languages such as MatLAB and C++, but with todays current best practices there is typically one simple function call in a modern machine learning framework that keeps track of all the gradients under the hood and implements this said algorithm. It also takes care of memory management, and can make the model run on GPU chips without much changes to the code. To me, I wanted to bring myself up to speed (to the best of my abilities) and applied for this course at Örebro university; a young university that has given me a very good impression.

I might need to add that I am also, according to some tests, green and red color blind (although I kindly beg to differ, and argue that I do in fact recognize the difference between these colors). It is perhaps a little bit ironic that I try to do this anyway, given my condition, but I should perhaps preface that maybe some of the results in this reports, where my own judgment is involved, may be a little bit haywire because of that.

2 Dataset

I chose to work with a dataset of landscape pictures found on the website Kaggle, published 4 years ago by the contributor "Arnaud Rougetet". A link to the dataset can be found here: https://www.kaggle.com/datasets/arnaud58/landscape-pictures. I downloaded this dataset on the 12 of february 2024. Email me a friendly request, and I can send you a copy of it.

The dataset consists of 4319 real-world photos, out of which there are 104 grey scaled images. The grey scaled images were filtered out before training. On the website I downloaded it from, the specification of the dataset was: landscapes (900 pictures), landscapes montain (900 pictures), landscapes desert (100 pictures), landscapes sea (500 pictures), landscapes beach (500 pictures), landscapes island (500 pictures), landscapes japan (900 pictures). Summing those numbers gives us 4300.

2.1 Rational behind dataset

Initially, my goal was ambitious and I worked with the COCO dataset from 2014 which contains images of all types of things. It was large, 82783 images in the training part. I noticed that it was also full of, for my project, contaminated gray scaled images. In order to produce models that worked well on all kinds of images, my intuition was that my network needed to increase its number of parameters drastically. My hardware could not give me that, and I ultimately decided to work with a much more reduced dataset focusing only on landscape pictures.

3 Data processing

3.1 Color space

A conventional way to display images, the format that most people are familiar with, is to quantify the image in red, green and blue color for each pixel. This is the RGB format. A grey scaled image, only has one dimension per pixel, and an RGB image has 3 dimensions per pixel. There is another way of encoding images known as the CIELAB color space. In CIELAB color space, the image is divided into three components, like RGB, but with the core difference that lightness, denoted **L***, is a separate component, set apart from the colors. Apart from the lightness component there are two components that together capture the four unique colors of human vision: red, green, blue and yellow, denoted as ***a** and ***b**. In CIELAB, a grey scale image can be produced by using the lightness component, and the colors can be produced by adding the ***a** and ***b** components. This color space makes it possible to create a model that predicts only two dimensions for all the colors and the lightness is preserved from the input image. I chose this format for the images to reduce the dimensionality of the data.

Figure 1: CIELAB color space, from left to right: Original image, L* channel only, L* and *a channels, L* and *b channels

3.1.1 Normalization

In CIELAB, the L* value takes its values in the interval from 0 to 100, while *a and *b goes from -127 to 127. I wanted to normalize my input so that these three variables were all in the interval -1 to 1 and did so with division by 127 in the *a and *b case and and division for 50 and substraction by one in the other case.

3.2 Code for loading image data

For the sake of documentation I will post code snippets here in this report. In the following section I will show you a class I made that handled the data processing part for my project.

3.2.1 Image loader in Python

Since the datasets I was working with was too large to have all in memory, I developed an image loader class that held a certain amount of them in memory and loaded in more as was needed. The "poll" function was used in the training loop to get the next batch of training data. I also used

the "shuffle" upon completing an epoch, that way I made the batches unique per epoch (training round).

```

from torchvision import transforms
import os
import pickle
import random
import sys
import numpy as np
import torch
from skimage.color import rgb2lab, lab2rgb
from PIL import Image
from torch.utils.data import DataLoader, Dataset

... OTHER CODE ...

class ImageLoader:
    def __init__(self, image_paths, device, dimension=(128,128), loaded_in_memory
    =1000):
        self.device = device
        self.image_paths = image_paths
        self.loaded_in_memory = loaded_in_memory
        self.data = []
        self.dim = dimension
        self.data_range = (0, 0)  # Start and end indices of images currently
    loaded in memory

    def image_to_lab_vector(self, image_path):
        image = Image.open(image_path).convert("RGB")
        resize_transform = transforms.Resize(self.dim)
        image = resize_transform(image)

        img = np.array(image)
        img_lab = rgb2lab(img).astype("float32") # Converting RGB to L*a*b

        img_lab[:, :, 0] = img_lab[:, :, 0] / 50 - 1 # Between -1 and 1 in L
        img_lab[:, :, 1] = img_lab[:, :, 1] / 127 # Between -1 and 1 in a
        img_lab[:, :, 2] = img_lab[:, :, 2] / 127 # Between -1 and 1 in b

        img_lab = img_lab.transpose((2, 0, 1))

        return img_lab

    def grey_scale_vector_to_file(self, image_greyscale_vector, file):
        image_rgb = ( image_greyscale_vector + 1 ) * 50
        l_channel = image_rgb[0, :, :].astype(np.uint8)

        # The l_channel now represents the grayscale image
        # If needed, you can convert this to a 3-channel grayscale image
        greyscale_image = np.stack((l_channel, l_channel, l_channel), axis=-1)
        image = Image.fromarray(greyscale_image)
        image.save(file)

    def l_and_ab_to_lab(self, l, ab):
        lab = torch.cat((l, ab), dim=1)
        return lab
```

```
53
54      def image_to_greyscale_vector(self, image_path):
55          image = Image.open(image_path).convert("RGB")
56          resize_transform = transforms.Resize(self.dim)
57          image = resize_transform(image)
58
59          img = np.array(image)
60          img_lab = rgb2lab(img).astype("float32") # Converting RGB to L*a*b
61
62          img_lab[:, :, 0] = img_lab[:, :, 0] / 50 - 1 # Between -1 and 1 in L
63          img_lab[:, :, 1] = img_lab[:, :, 1] / 127 # Between -1 and 1 in a
64          img_lab[:, :, 2] = img_lab[:, :, 2] / 127 # Between -1 and 1 in b
65
66          img_lab = img_lab.transpose((2, 0, 1))
67
68          # Convert image to vector
69          image_vector = img_lab
70
71          # Create a greyscale image version
72          image_greyscale = img_lab[0, :, :]
73
74          image_greyscale_vector = np.array(image_greyscale)[np.newaxis, ...]
75
76          return image_greyscale_vector
77
78      def image_vector_to_file(self, image_vector_lab, file):
79          if isinstance(image_vector_lab, torch.Tensor):
80              image_vector_lab = image_vector_lab.detach().numpy()
81
82          image_vector_lab[0, :, :] = (image_vector_lab[0, :, :] + 1) * 50
83
84          image_vector_lab[1, :, :] = image_vector_lab[1, :, :] * 127
85          image_vector_lab[2, :, :] = image_vector_lab[2, :, :] * 127
86
87          # Clipping the LAB values to a valid range
88          image_vector_lab[0, :, :] = np.clip(image_vector_lab[0, :, :], 0, 100)
89          image_vector_lab[1, :, :] = np.clip(image_vector_lab[1, :, :], -128, 127)
90          image_vector_lab[2, :, :] = np.clip(image_vector_lab[2, :, :], -128, 127)
91
92          image_lab = image_vector_lab.transpose((1, 2, 0))
93          image_rgb = lab2rgb(image_lab) * 255
94
95          image = Image.fromarray(image_rgb.astype(np.uint8))
96          image.save(file)
97
98      def load_images(self, start_index, end_index):
99          self.data = []
100         for i in range(start_index, min(end_index, len(self.image_paths))):
101             image_path = self.image_paths[i]
102             image = Image.open(self.image_paths[i]).convert("RGB")
103             resize_transform = transforms.Resize(self.dim)
104             image = resize_transform(image)
105
106             img = np.array(image)
107             img_lab = rgb2lab(img).astype("float32") # Converting RGB to L*a*b
108
```

```
            img_lab[:, :, 0] = img_lab[:, :, 0] / 50 - 1 # Between -1 and 1 in L
            img_lab[:, :, 1] = img_lab[:, :, 1] / 127 # Between -1 and 1 in a
            img_lab[:, :, 2] = img_lab[:, :, 2] / 127 # Between -1 and 1 in b

            img_lab = img_lab.transpose((2, 0, 1))

            # Convert image to vector
            image_vector = img_lab

            # Create a greyscale image version
            image_greyscale = img_lab[0, :, :]

            image_greyscale_vector = np.array(image_greyscale)[np.newaxis, ...]

            self.data.append((image_greyscale_vector, image_vector, image_path))

        self.data_range = (start_index, min(end_index, len(self.image_paths)))

    def poll(self, batch_index, batch_size):
        start_index = batch_index * batch_size
        end_index = start_index + batch_size

        # Check if batch is within the range of currently loaded images
        if not (self.data_range[0] <= start_index < self.data_range[1] and self.
    data_range[0] <= end_index <= self.data_range[1]):
            self.load_images(start_index, end_index)

        selected_data = self.data[start_index - self.data_range[0] : end_index -
    self.data_range[0]]

        nbr_images = len(selected_data)
        if nbr_images == 0:
            return None

        train_x = np.zeros((nbr_images, 1, self.dim[1], self.dim[0]))
        train_y = np.zeros((nbr_images, 2, self.dim[1], self.dim[0]))

        for i, (example_x, example_y, image) in enumerate(selected_data):
            train_x[i] = example_x
            train_y[i] = example_y[1:, :, :]

        return (
            torch.from_numpy(train_x).float().to(self.device),
            torch.from_numpy(train_y).float().to(self.device),
            nbr_images,
        )

    def shuffle(self):
        random.shuffle(self.image_paths)
        self.data = []
        self.data_range = (0, 0)
```

Listing 1: ImageLoader class

3.3 Image size

Initially, when I experimented with model architectures, training and other procedures, I worked with small images of 128 pixels width and height (128x128). When I eventually felt that some kind of progress had been made, I worked with 256 width and 256 height pixels instead (256x256). Images present in the dataset of another size were made to fit that size using bilinear interpolation.

4 Modelling

The problem will consist of developing a program that generates images with color using deep learning. I will call the active generating part of this program a generator network. Reading up on the problem online I found a paper from KTH where a masters thesis student had done this problem by implementing a generator network based on a so called UNet architecture and training this using an adversarial approach, see Deng 2023.

I wanted to copy that and build an image colorizer that consists of a generator network, that is trained using an adversarial mechanism in which a discriminator network plays an important role.

I chose to work with PyTorch in Python as the tool for developing the program.

4.1 Network structure

For the generator network, I wanted to implement a so-called UNet architecture. For 256x256 images, this network takes input tensors of dimension (X, 256, 256, 1) and outputs (X, 256, 256, 2) where X denotes the batch size. This network takes the lightness of each pixel and predicts the a* and b* for each pixel. Concatenating these two, creates a full image in CIELAB color space. For the discriminator network, I wanted a network that was not as deep as the generator that takes the input of (X, 256, 256, 1) sized tensors, and outputs a value between 0 and 1 on the form (X, 1). This will be a binary classifier and the architecture will also be based on the UNet architecture.

4.1.1 Generator Network

The generator network is a UNet network. This architecture apparently does a good job of segmenting image data and seems to be the "state of the art" type of architecture for a variety of tasks in computer vision. One can implement this in many ways, but the basic rule is that this network performs autoencoding, in which the data dimension is conctracting to a lower dimensionality and then expanda back into its original dimension.

In order to scale the model size, I introduced the hyperparameter called "additional_complexity" which was used at all levels to scale the model horizontally.

Here is the code I wrote for this network:

```
1 import torch
2 import torch.nn as nn
3 import torch.nn.functional as F
4
5 class UNetBlock(nn.Module):
6     def __init__(self, channels_in, channels_out, downsampling=False, upsampling=
    False, lastLayer=False):
7         super().__init__()
8
```

```
        downconv = nn.Conv2d(channels_in, channels_out, kernel_size=4, stride=2, padding=1, bias=False)
        downrelu = nn.LeakyReLU(0.2)
        downnorm = nn.BatchNorm2d(channels_out)

        uprelu = nn.ReLU()
        upnorm = nn.BatchNorm2d(channels_out)
        upconv = nn.ConvTranspose2d(channels_in, channels_out, kernel_size=4, stride=2, padding=1, bias=False)

        if downsampling:
            model = [downconv, downrelu, downnorm]
            self.model = nn.Sequential(*model)
        elif upsampling:
            model = [uprelu, upconv, upnorm, nn.Dropout(0.5)]
            self.model = nn.Sequential(*model)
        elif lastLayer:
            model = [uprelu, upconv, nn.Tanh()]
            self.model = nn.Sequential(*model)
        else:
            raise ValueError("The UNet block must either be downsampling or upsamplig, or be the last layer")

    def forward(self, x):
        return self.model(x)


class UNet(nn.Module):
    def __init__(self, n_channels, n_classes, complexity=30):
        super(UNet, self).__init__()
        self.n_channels = n_channels
        self.n_classes = n_classes

        self.additional_complexity = complexity

        self.inc = nn.Sequential(*[
            nn.Conv2d(n_channels, self.additional_complexity, kernel_size=3, stride=1, padding=1, bias=False),
            nn.LeakyReLU(0.2)
        ])

        self.down1 = UNetBlock(self.additional_complexity, self.additional_complexity, downsampling = True)
        self.down2 = UNetBlock(self.additional_complexity, self.additional_complexity, downsampling = True)
        self.down3 = UNetBlock(self.additional_complexity, self.additional_complexity, downsampling = True)
        self.down4 = UNetBlock(self.additional_complexity, self.additional_complexity, downsampling = True)
        self.down5 = UNetBlock(self.additional_complexity, self.additional_complexity, downsampling = True)

        self.up1 = UNetBlock(self.additional_complexity, self.additional_complexity, upsampling = True)
        self.up2 = UNetBlock(2*self.additional_complexity, self.additional_complexity, upsampling = True)
```

```
54          self.up3 = UNetBlock(2*self.additional_complexity, self.
    additional_complexity, upsampling = True)
55          self.up4 = UNetBlock(2*self.additional_complexity, self.
    additional_complexity, upsampling = True)
56          self.up5 = UNetBlock(2*self.additional_complexity, self.
    additional_complexity, lastLayer = True)
57
58          self.outc = nn.Sequential(*[
59              nn.Conv2d(self.additional_complexity, self.n_classes, kernel_size=3,
    stride=1, padding=1, bias=False),
60              nn.Tanh()
61          ])
62
63          self.debug = False#True
64
65      def forward(self, x):
66
67          x = self.inc(x)
68
69          x1 = self.down1(x)
70          x2 = self.down2(x1)
71          x3 = self.down3(x2)
72          x4 = self.down4(x3)
73          x5 = self.down5(x4)
74
75          u1 = self.up1(x5)
76          u2 = self.up2(torch.cat([u1, x4], dim=1))
77          u3 = self.up3(torch.cat([u2, x3], dim=1))
78          u4 = self.up4(torch.cat([u3, x2], dim=1))
79          u5 = self.up5(torch.cat([u4, x1], dim=1))
80
81          x = self.outc(u5)
82
83          return x
```

Listing 2: UNet class

Dimensionality reduction is performed by using a Conv2d layer with stride set to 2. This effectively reduces the dimension by a half. It does this 5 times, so for images of size 256, the last dimension will be $\frac{256}{2^5} = 8$. Dimensionality expansion is done with the ConvTranspose2d that "undoes" this dimensionality reduction. The flow of data is showed most in the "forward" function. The final layer consists of a "tanh" activation that makes the output fit the -1 and 1 interval. I used ReLU as activation function in the other cases. I also used BatchNorm that tries to normalize the gradients, preventing them from growing to large or vanishing, which is needed when training deep networks. I also used Dropout for the upsampling layers. The Dropout is a way to build an ensamble effect on the network, a form of regularization technique that hopefully increases the generalization ability of the model.

4.1.2 Discriminator Network

The discriminator network was implemented using the UNetBlock class developed for the generator, showed above. It was implemented with a UNet structure, but it was not as deep as the generator.

In order to scale the model size, I introduced the hyperparameter "additional_complexity" which was used at all levels to scale the model horizontally.

Here is the code I wrote for this network:

```
class Discriminator(nn.Module):
    def __init__(self, n_channels=3, complexity=30, dimension=128):
        super(Discriminator, self).__init__()

        self.n_channels = n_channels

        self.additional_complexity = complexity

        self.inc = nn.Sequential(*[
            nn.Conv2d(n_channels, self.additional_complexity, kernel_size=3, stride=1, padding=1, bias=False),
            nn.LeakyReLU(0.2)
        ])

        self.down1 = UNetBlock(self.additional_complexity, self.additional_complexity, downsampling = True)
        self.down2 = UNetBlock(self.additional_complexity, self.additional_complexity, downsampling = True)
        self.down3 = UNetBlock(self.additional_complexity, self.additional_complexity, downsampling = True)

        self.up1 = UNetBlock(self.additional_complexity, self.additional_complexity, upsampling = True)
        self.up2 = UNetBlock(2*self.additional_complexity, self.additional_complexity, upsampling = True)
        self.up3 = UNetBlock(2*self.additional_complexity, self.additional_complexity, lastLayer = True)

        # Final fully connected layer to get a single output
        self.fc = nn.Sequential(
            nn.Flatten(),
            nn.Linear(self.additional_complexity * dimension * dimension, 1),
            #nn.BatchNorm1d(1),
        )

        self.activation = nn.Sigmoid()

    def forward(self, x):
        x = self.inc(x)
        #print(x[0][0])

        x1 = self.down1(x)
        x2 = self.down2(x1)
        x3 = self.down3(x2)

        u1 = self.up1(x3)
        u2 = self.up2(torch.cat([u1, x2], dim=1))
        u3 = self.up3(torch.cat([u2, x1], dim=1))

        # Flatten and pass through the fully connected layer
        x = self.fc(u3)
        x = x / self.additional_complexity

```

```
48          return self.activation(x)
```

Listing 3: Discriminator class

Instead of the generator down sampling that is done 5 times, this is done only 3 times. I thought that it might be enough. But I could also be wrong. This was the network structure I ended up using. So for images of size 256, the last dimension will be $\frac{256}{2^3} = 32$. I noticed that if I scaled up my network, the output would be something extreme like 0 or 1. I therefore added the normalization on line 46. Purely practical, I am sure there is something I am missing, but this turned out good. Also, I tried to use a BatchNorm at the end also, but then my output would trend towards 0.5 which made the discriminator useless.

4.1.3 Generative Adversarial Network

A generative adversarial network consists of a generator and a discriminator model that optimize for different objectives and together help boost the resemblance of the generator predictions with the original content in the training set. The goal of this generative network is to predict colors that resemble what is present in the dataset. We want the resemblance of the model output with its original counterpart to be high. If we train a predictor network to distinguish between original images and generated images during the training, we want the predictor to inform the generator of when that resemblance is lost, so that the generator can optimize for the quality of resemblance and we do this by optimizing to "fool" the predictor during training. This is the intuition of an adversarial training procedure. The generator will not only try to optimize for the average output difference between the generator output and original image (with for example L1 or L2 norm loss), it will also optimize for resemblance by "fooling" a predictor that is trying to distinguish between original and generative data (with a binary cross entropy loss). This adversarial nature of the training procedure make the training loop look a little unconventional compared to standard model training and there will be separate metrics taken for the process measuring how well a training process is going.

4.1.4 Reconstruction and adversarial loss

To measure how well, on average, the generator network outputs differ from the ones present in the training set, I apply a L1 norm loss to the difference between the output and the original. It is a unitless value that I call **reconstruction loss**, and it measures the following:

$$\text{Reconstruction Loss} = \frac{1}{N}\sum_{i=1}^{N}|x_i - y_i|$$

where N is the batch size, x is model color prediction and y is original image color. The adversarial loss is also computed, as a binary cross entropy, which is a unitless value that measure how well the model predicts what is original and what is generated, the **discriminator loss**, which is defined as the following:

$$\text{Discriminator Loss} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\right)$$

where N is the batch size, p is the model prediction, y is 1 if the input data was from the training dataset and 0 if the input data was from a generated image. When the generator tries

to "fool" the discriminator, we flip the "truth" of the prediction. We define the **adversarial loss** like we fefine the reconstruction loss, except that the value of y is actually flipped. So y is 1 for the generator output example.

$$\text{Adversarial Loss} = -\frac{1}{N}\sum_{i=1}^{N}(\log(p_i))$$

where p_i is the discriminator prediction for the generator output of index i in the batch.

4.2 Training in an adversarial manner

Here is the code I wrote for training the Discriminator and Generator networks in an adversarial manner, implemented with the help of PyTorch in Python:

```
... IMPORTS ...

# Training on a batch of data
def train(
    generator, discriminator, batch, optimizer_g, optimizer_d, criterion_generator,
    criterion_discriminator, iterations_acc=0, learningrate=0.003,
    train_generator = True, train_discriminator = True, adversarial_factor = 0.1,
   image_loader = None
):
    train_x, train_y = batch

    xy_combined = torch.cat([train_x, train_y], dim=1)

    device = next(generator.parameters()).device

    optimizer_g.zero_grad()
    optimizer_d.zero_grad()

    # Train discriminator on real image
    real_labels = torch.zeros(train_y.size(0)).to(device)
    real_labels += 1
    output = discriminator(xy_combined).view(-1)
    loss_discriminator_real = criterion_discriminator(output, real_labels)
    if train_discriminator:
        loss_discriminator_real.backward()
        optimizer_d.step()
    # Train generator, store output for fake image
    fake_image = generator(train_x).detach()

    # Train discriminator on fake image
    fake_labels = torch.zeros(train_y.size(0)).to(device)
    x_fake_combined = torch.cat([train_x, fake_image], dim=1)
    d_output = discriminator(x_fake_combined).view(-1)
    loss_discriminator_fake = criterion_discriminator(d_output, fake_labels)
    if train_discriminator:
        loss_discriminator_fake.backward()
        optimizer_d.step()

    optimizer_g.zero_grad()
    optimizer_d.zero_grad()

```

```
    # Generate output from grey scale image
    output = generator(train_x)
    # How far off is the generator from the original image
    content_loss = criterion_generator(output, train_y)
    # What does the discriminator say about the generated image
    output_combined = torch.cat([train_x, output], dim=1)
    output_disc = discriminator(output_combined)

    # I want to fool the discriminator to say that it is true (real)
    input_label = torch.zeros(train_x.size(0), 1).to(device)
    input_label += 1
    advesarial_loss = criterion_discriminator(output_disc, input_label)
    if train_generator:
        # scale the loss of the discriminator
        loss_generator = (1-adversarial_factor) * content_loss + adversarial_factor
    * advesarial_loss
        loss_generator.backward()

        optimizer_g.step()

    return content_loss.item(), loss_discriminator_real.item(), advesarial_loss.
   item(), loss_generator.item(), output_disc

... SOME CODE ...

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    ... DEFINING SOME INPUT ARGUMENTS ...
    args = parser.parse_args()

    batch_size = args.batch_size
    device = args.device
    loss_mov_avg = args.loss_mov_avg
    no_disc = args.no_disc
    complexity = args.complexity
    disc_complexity = args.disc_complexity
    losses_file = args.losses_file
    store_net = args.store_net
    img_scale = args.img_scale
    step_decay = args.step_decay
    step_decay_factor = args.step_decay_factor
    adversarial_factor = args.adversarial_factor
    dimension = (args.dim, args.dim)
    max_epocs = args.epochs
    file_gen = args.model_gen
    file_disc = args.model_disc

    a = UNet(1, 2, complexity=complexity)
    d = Discriminator(3, complexity=disc_complexity, dimension=dimension[0])

    # Using mps if training on my mac
    if torch.backends.mps.is_available() and device == "mps":
        print("mps available")
        a = a.to("mps")
        d = d.to("mps")
    else:
```

```
        a = a.to(device)
        d = d.to(device)

    images = list_images(args.images)
    print(f"Found {len(images)} images under {args.images}.")

    optimizer = optim.Adam(a.parameters(), lr=args.lr)
    criterion_generator = torch.nn.L1Loss()
    criterion_discriminator = torch.nn.BCELoss()
    optimizer_d = optim.Adam(d.parameters(), lr=args.lr_disc)

    batch_load = args.batch_load
    disc_hold_out = args.disc_hold_out
    batch_count = 0
    iterations = 0
    nbr_images_in_data = len(images)

    image_loader = ImageLoader(images, device, dimension=dimension,
    loaded_in_memory=batch_load)

    print(f"Will process {batch_load} images at a time in memory")
    total_params_gen = sum(p.numel() for p in a.parameters())
    total_params_disc = sum(p.numel() for p in d.parameters())
    print(f"Generator number of parameters:     {total_params_gen}")
    print(f"Discriminator number of parameters: {total_params_disc}")
    print(f"Adversarial loss factor: {adversarial_factor}")
    print(f"Discrimator hold out: {disc_hold_out}")

    loss_generator_mov_avg = None
    loss_discriminator_mov_avg = None
    loss_adversarial_mov_avg = None
    loss_total_mov_avg = None
    batch_count = 0
    epochs = 0
    loss_content = ""
    while iterations < args.itr and epochs <= max_epocs:
        loss = 0
        # train discriminator at this rate
        discriminator_train_rate = disc_hold_out # every forth
        batch_walk = 0

        if batch_count * batch_size >= nbr_images_in_data:
            epochs += 1
            batch_count = 0
            image_loader.shuffle()
            continue

        train_x, train_y, nbr_examples = image_loader.poll(batch_count, batch_size)

        if nbr_examples != batch_size:
            epochs += 1
            batch_count = 0
            image_loader.shuffle()
            continue

        train_x = train_x.to(device)
```

```
149         train_y = train_y.to(device)
150 
151         # Create batches for train_x and train_y
152         train_x_batches = torch.split(train_x, batch_size)
153         train_y_batches = torch.split(train_y, batch_size)
154 
155         # Store status to csv
156         csv_filename = losses_file
157         loss_file = open(csv_filename, mode='a', newline='')
158 
159         disc_output = [0]
160         while batch_walk < len(train_x_batches) and iterations < args.itr:
161             train_discriminator = (iterations % discriminator_train_rate) == 0
162             if no_disc:
163                 train_discriminator = False
164             content_loss, loss_discriminator, adversarial_loss, total_loss, disc_output = train(
165                 a,
166                 d,
167                 (train_x_batches[batch_walk], train_y_batches[batch_walk]),
168                 optimizer,
169                 optimizer_d,
170                 criterion_generator,
171                 criterion_discriminator,
172                 iterations,
173                 train_discriminator = train_discriminator,
174                 train_generator = True,
175                 adversarial_factor = adversarial_factor,
176                 image_loader = image_loader
177             )
178             iterations += 1
179             batch_count += 1
180             batch_walk += 1
181 
182             if loss_discriminator_mov_avg == None:
183                 loss_discriminator_mov_avg = loss_discriminator
184                 loss_generator_mov_avg = content_loss
185                 loss_adversarial_mov_avg = adversarial_loss
186                 loss_total_mov_avg = total_loss
187             else:
188                 loss_discriminator_mov_avg = loss_discriminator_mov_avg * loss_mov_avg + (1.0 - loss_mov_avg) * loss_discriminator
189                 loss_generator_mov_avg = loss_generator_mov_avg * loss_mov_avg + (1.0 - loss_mov_avg) * content_loss
190                 loss_adversarial_mov_avg = loss_adversarial_mov_avg * loss_mov_avg + (1.0 - loss_mov_avg) * adversarial_loss
191                 loss_total_mov_avg = loss_total_mov_avg * loss_mov_avg + (1.0 - loss_mov_avg) * total_loss
192 
193             generator_learning_rate = optimizer.param_groups[0]['lr']
194             discriminator_learning_rate = optimizer_d.param_groups[0]['lr']
```

Listing 4: Adversarial training

The lines 41 to 51 in the code snippet above, show how we "fool" the discriminator. We have taken output from the generator and fed it into the discriminator (line 41 to 46), we then calculate a loss as if it was "real" and use these gradients for training the generator network. In other words,

we optimize the generator to make the discriminator say that it is real. This is how we optimize for resemblance. With the "adverarial_factor", I can alter how much influence the GAN effect will have on the network, the results on that will be presented.

4.2.1 Model evaluation

Ultimately, I want a model that can colorize images to a satisfying degree. This is subjective territory. But to evaluate model performance, I will look at the loss values to guide me through the process of setting hyperparameters and I will ultimately decide what models I prefer over others based on how satisfied I am personally by the way they colorize grey scaled photos. These gray scaled photos are taken from the internet, not present in the model training datasets, and scraped from the web by me.

5 Results

The model architecture was chosen rather early on, but I did experiment a lot. However, most of my experiments were not fruitful and did not generate anything interesting. Deciding on switching from RGB to CIELAB had a huge impact on model overall performance, but since that was so early on, I did not store any metrics supporting that claim. Most of the results presented here were gather after I have proved that I could in fact produce generally satisfying results. And I took a deep dive in exploring the hyperparameters impact on the overall performance as well as deciding for myself what the end result would be presented like, how much resemblance I could maximally get. I am sure, that given more time and compute; I could have presented an even more satisfying end result; but that will be up to me to prove in the future.

After having produced the model architecture in PyTorch and having started to get results that weren't all bad, I took time to evaluate what to set the hyperparameters to, most notably the additional_complexity and the adversarial_factor hyperparameters.

5.1 Adversarial effect

If adversarial effect really needed? I tested this hypothesis. With the hyperparameter **adversarial_factor** set to 0 and 0.9 respectively, for a given set of other hyperparameters, I gathered the results that the generator outputted mostly brown or grey scale variants of the output. I think this is because it learned to reduce the reconstruction loss by simply averaging the values of the channels, resulting in a very mono-colored image. In fact, setting the adversarial factor to zero made the reconstruction loss very small, it produced the minimum reconstruction loss I reached. But it increased the adversarial loss over time, which hinted that some level of resemblance wasn't achieved. You can see and compare the images I gathered:

Figure 2: Model output for model trained without GAN, 3x6 grid: original images in the left 3x3 grid, output images in the right 3x3 grid. Generator additional complexity set to 46 (476054 parameters), discriminator to 16 (1081937 parameters).

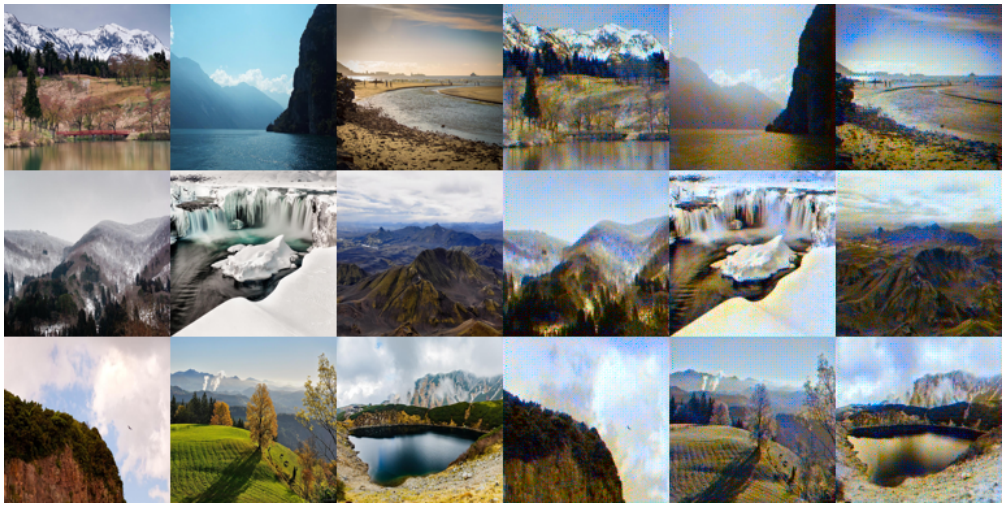

Figure 3: Model output for model trained with a high GAN factor (0.9), 3x6 grid: original images in the left 3x3 grid, output images in the right 3x3 grid. Generator additional complexity set to 46 (476054 parameters), discriminator to 16 (1081937 parameters).

Setting the GAN factor high, generated output with good choices for colors, the blue sky, the sand is beige and the trees are all green. The color intensity is also on point. However, when looking closely on the pictures, they were very strangely pixelated. Setting it to zero resulted in mono-colorization which was not what we were trying to achieve. I experimented further on with a non-zero adversarial_factor hyperparameter to find the in my opinion best trade off between smoothness of colors and intensity. I studied the reconstruction loss, discriminator loss (how well it differentiates between fake and real) as well as the adversarial loss (how well the generator is fooling the discriminator) during the training process. The results were visualized with a Python library called "plotly". I took some screen shots of those reports and they are shown in the figures below. The generator additional complexity hyperparameter was set to 46 resulting in 476054 parameters and the discriminator additional complexity hyperparameter was set to 16 resulting in 1081937 parameters.

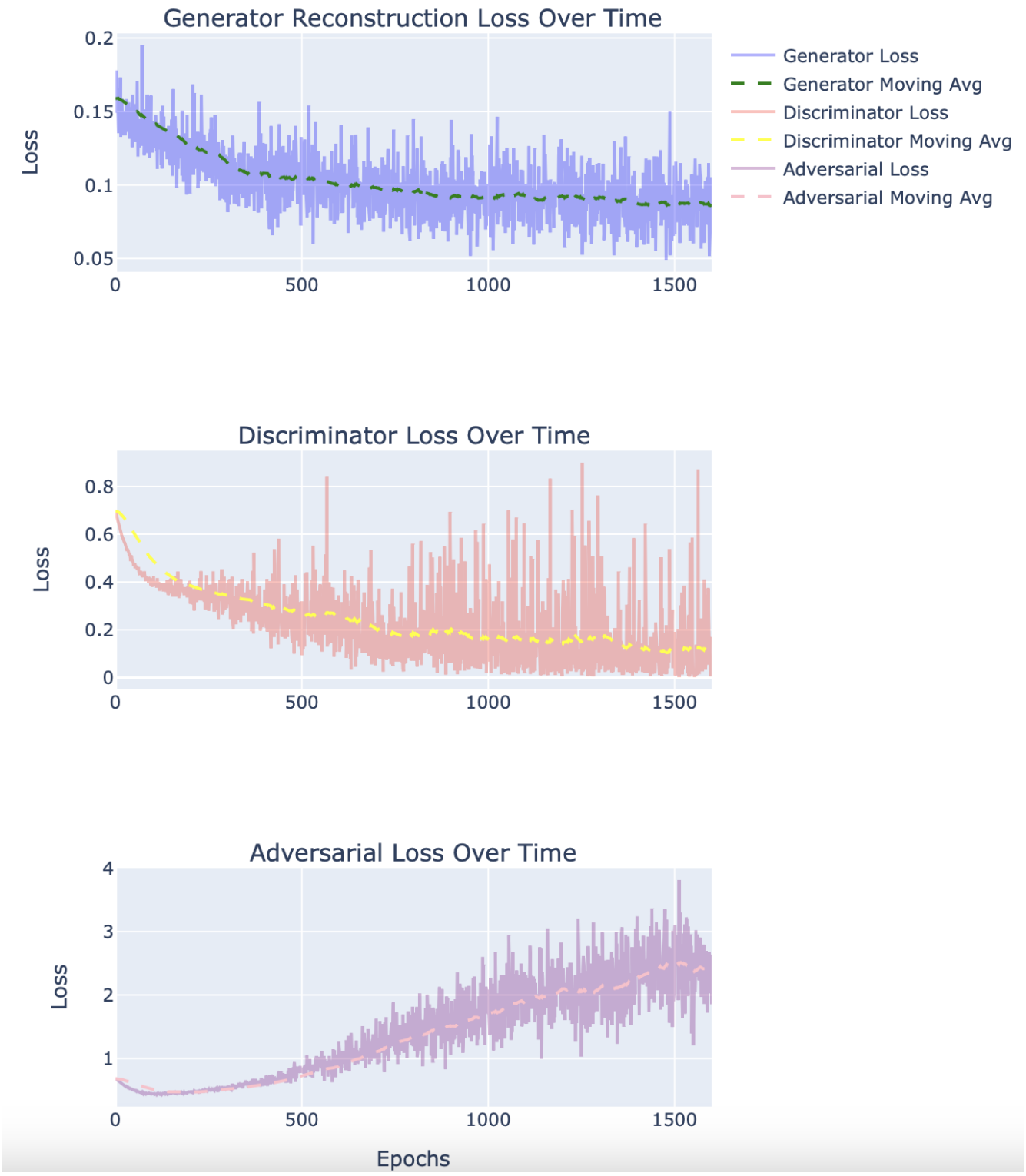

Figure 4: Loss of the model during training without any GAN effect. Generator additional complexity set to 46 (476054 parameters), discriminator to 16 (1081937 parameters).

When the adversarial factor is zero, the adversarial loss grows steadily as seen in 4. This means that the predictor learns to differentiate between a real and a generated image, and that could mean that the generator fails to resemble what it is intended to resemble in the eyes of a human observer. However, for a successful training run, I want the generator reconstruction loss to go down while not loosing the resemblance and this is achieved in the latter case as seen in . After this type of experimentation, I decided to set this hyperparameter to:

$$\text{adversarial\_factor} = 0.1$$

5.2 Model complexity

How large does the model need to be? I tested training networks with different sizes by tweaking the additional_complexity hyperparameter. I decided to freeze the model complexity of the discriminator network, and only alter the complexity for the generator as I checked the consequence of increasing model complexity. I set the batch size to 16, adversarial factor to 0.1, learning rate to 7e-6, discriminator complexity 4 (rendering a model with 264341 parameters). I trained for 14 epochs generator model with 16, 24, 30, 38, 46, 52. I learned that the adversarial loss steadily increased for the 16 and 24 sized models, while stagnated at 30 and beyond. As the model size increased, the training runs ran slower and slower. With the stagnated reconstruction loss, the discriminator. When the adversarial loss decreases, the output tends to be more mono-colored and the image color intensity is reduced. I therefore want a matching discriminator and generator network.

I thought about how to report these results, and just providing the final moving average loss values would not be very wise since I think they fluctuate too much between runs. This is expected as the weights of the networks are randomly initialized. A scientific way would be to make many runs, and then compute the mean and variance for each loss value (reconstruction and adversarial) but that would take a long time for me to do. So instead I will just show you the figures of the reconstruction and adversarial loss during training changed as the model grew in size.
As the model complexity grows, we can see how the discriminator network goes from being "stronger" than the generator, as indicated by a steady growth of the adversarial loss, to the generator being "stronger" than the discriminator network, as indicated by a rocky decrease of the adversarial loss.

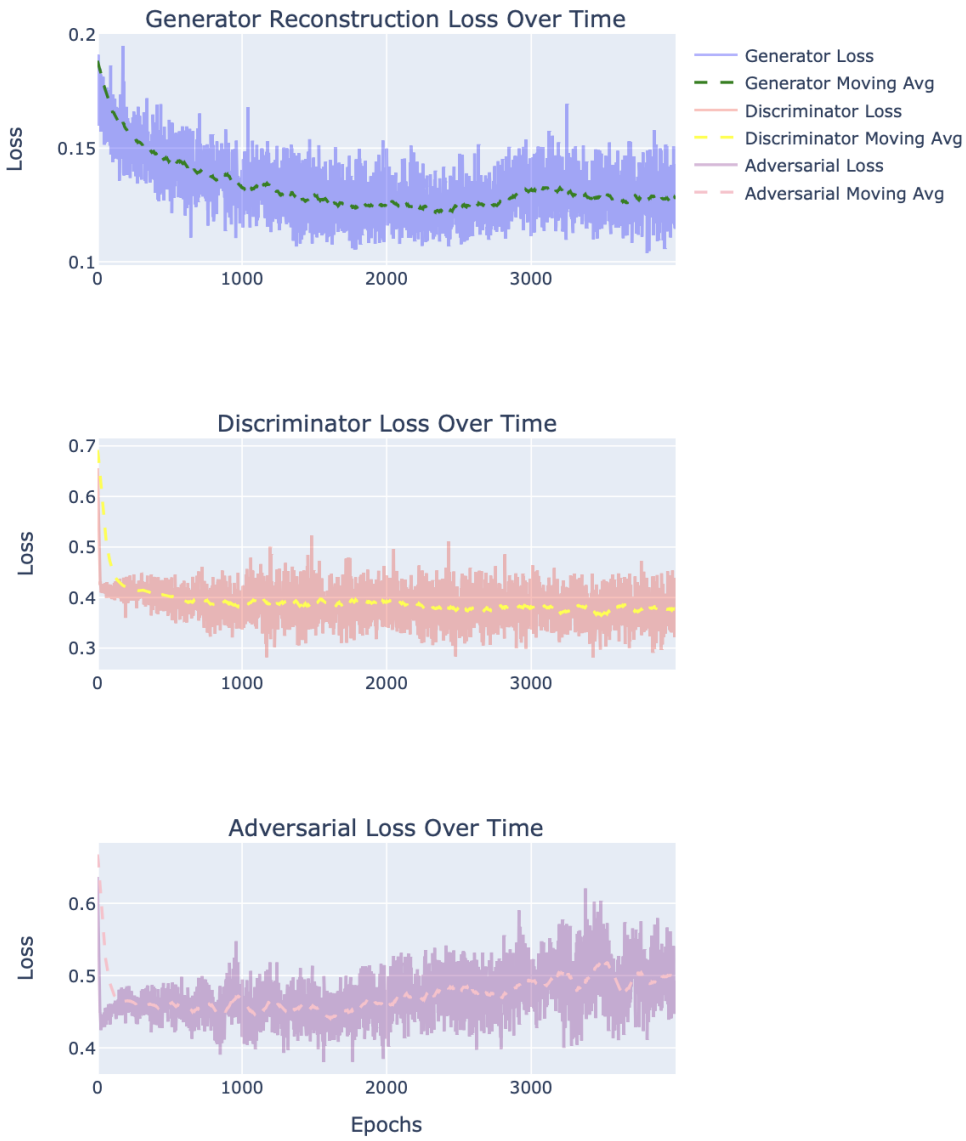

Figure 5: Loss of the model during training generator model complexity hyperparameter set to 16 and discriminator model complexity set to 4. Adversarial loss rises steadily.

In 5 the discriminator model has 264341 parameters, and the generator network has 58064 parameters. We can see how the adversarial loss rises steadily, this suggest that the generator network is no match for the discriminator to recognize. The generator increasingly fails to "fool" the discriminator and the discriminator learns to recognize the difference between the generator output and the originals from the training dataset.

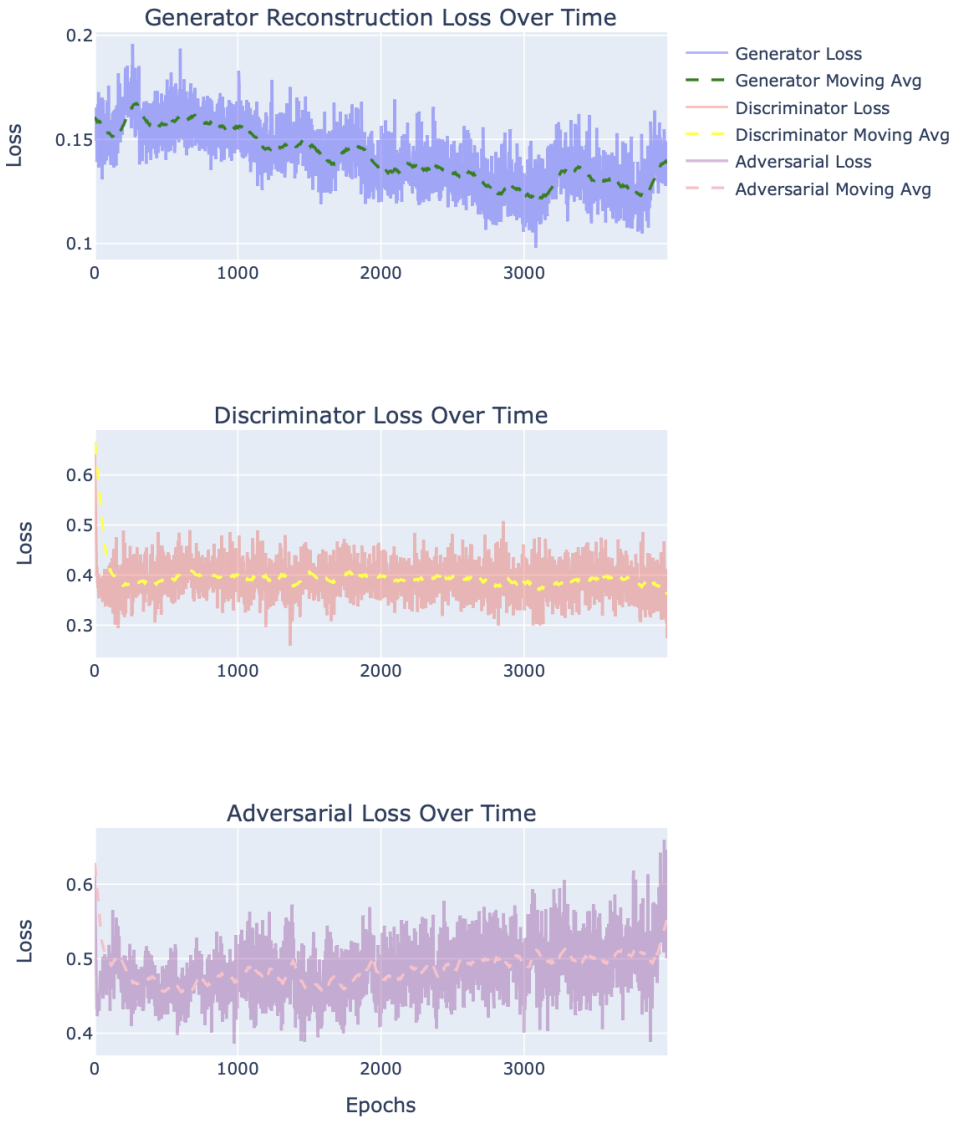

Figure 6: Loss of the model during training generator model complexity hyperparameter set to 24 and discriminator model complexity set to 4. Adversarial loss rises slowly.

In 6 the discriminator model has 264341 parameters, and the generator network has 130104 parameters. We can see how the adversarial loss rises but will less intensity as shown in 5 when the complexity hyperparameter was set to 16.

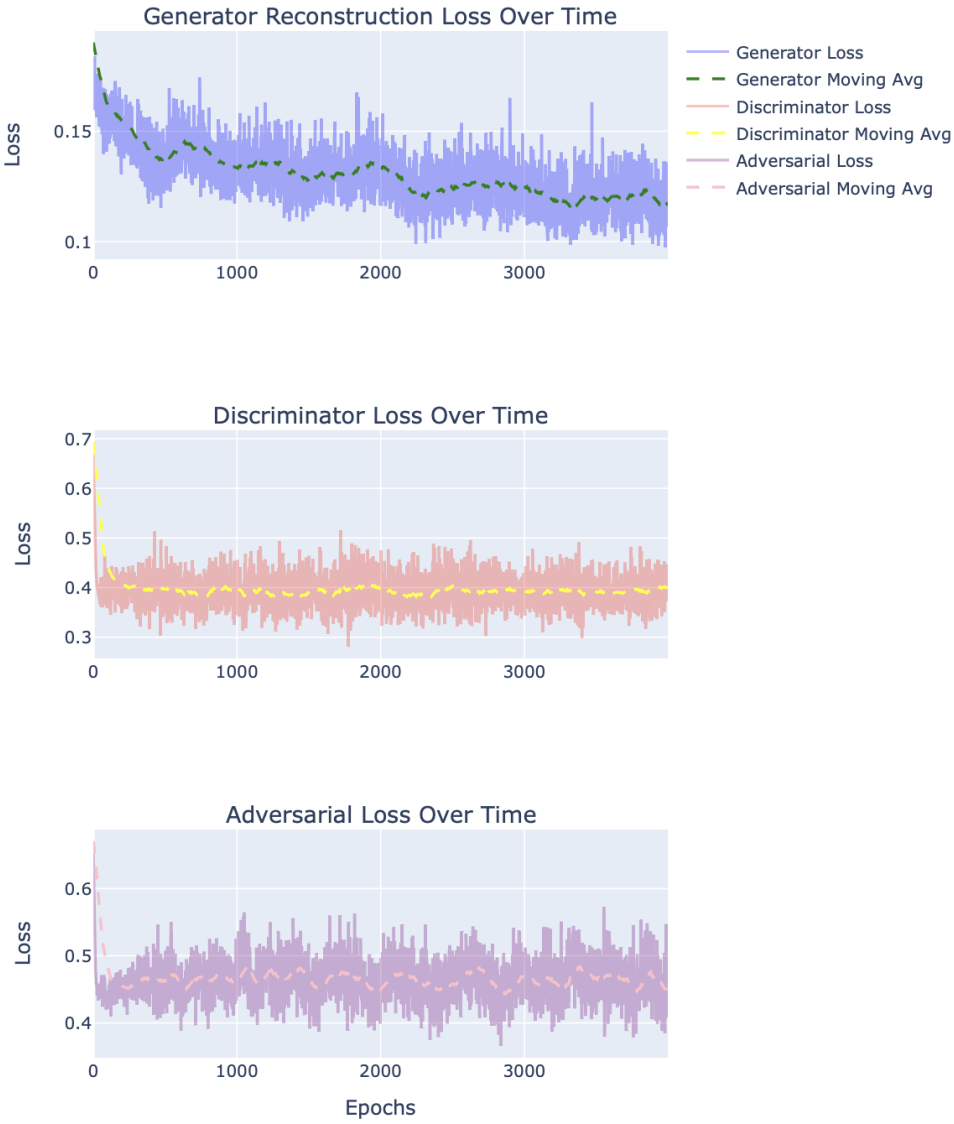

Figure 7: Loss of the model during training generator model complexity hyperparameter set to 30 and discriminator model complexity set to 4.

In 7 the discriminator model has 264341 parameters, and the generator network has 202950 parameters. Adversarial loss barely raises, almost stagnant. The generator almost matches the discriminator.

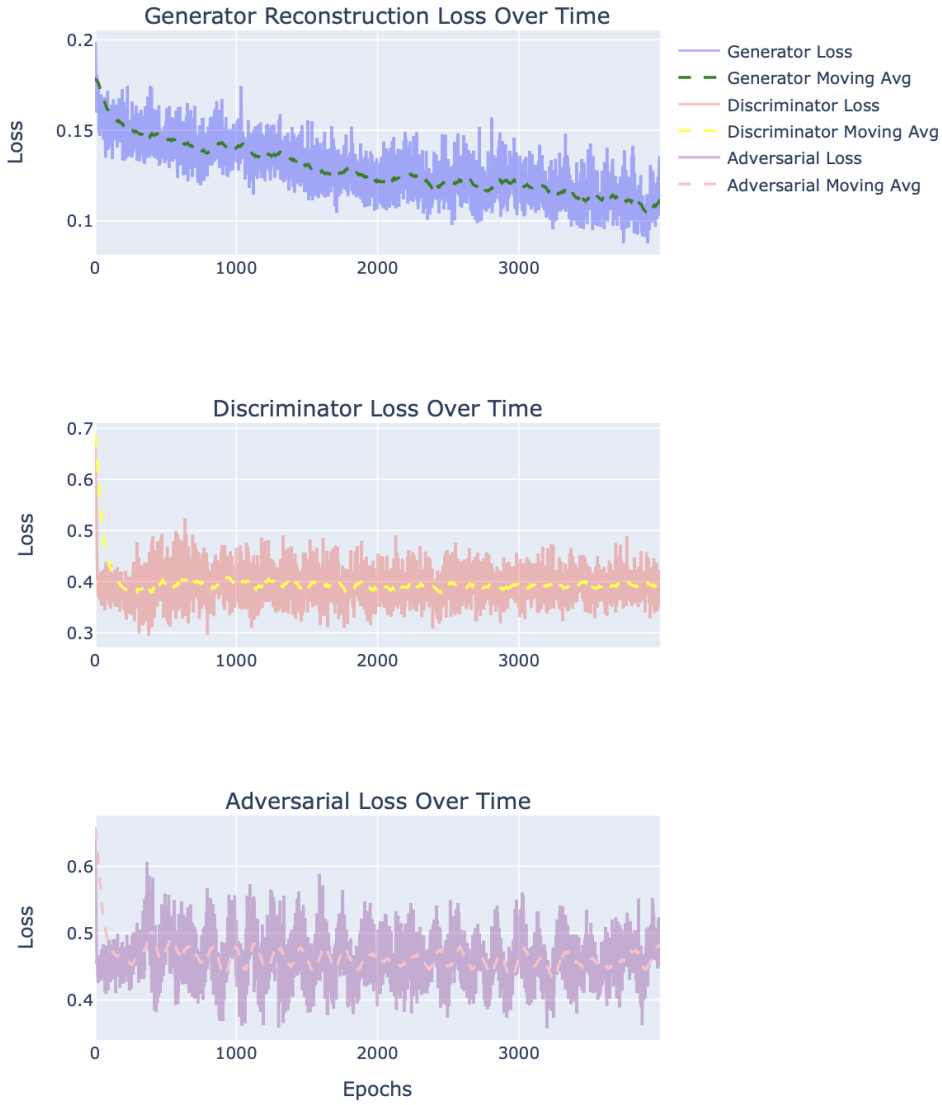

Figure 8: Loss of the model during training generator model complexity hyperparameter set to 38 and discriminator model complexity set to 4.

In 8 the discriminator model has 264341 parameters, and the generator network has 325166 parameters. The ratio of discriminator network to generator model, $\frac{264341}{325166}$ is approximately 0.8. My intuition is that I want to keep this ratio if I want to scale up the generator network, which I do since I am not sufficiently happy with the final output. The generator matches the discriminator during the training phase. This is a good sign when traning GANs, this means that the training can be effective. We see that the reconstruction loss is going steadily down. I could test this configuration more, and train for a longer period of epochs until the reconstruction loss stagnates at another value. But my experience tells me that this will stagnate at around 0.10.

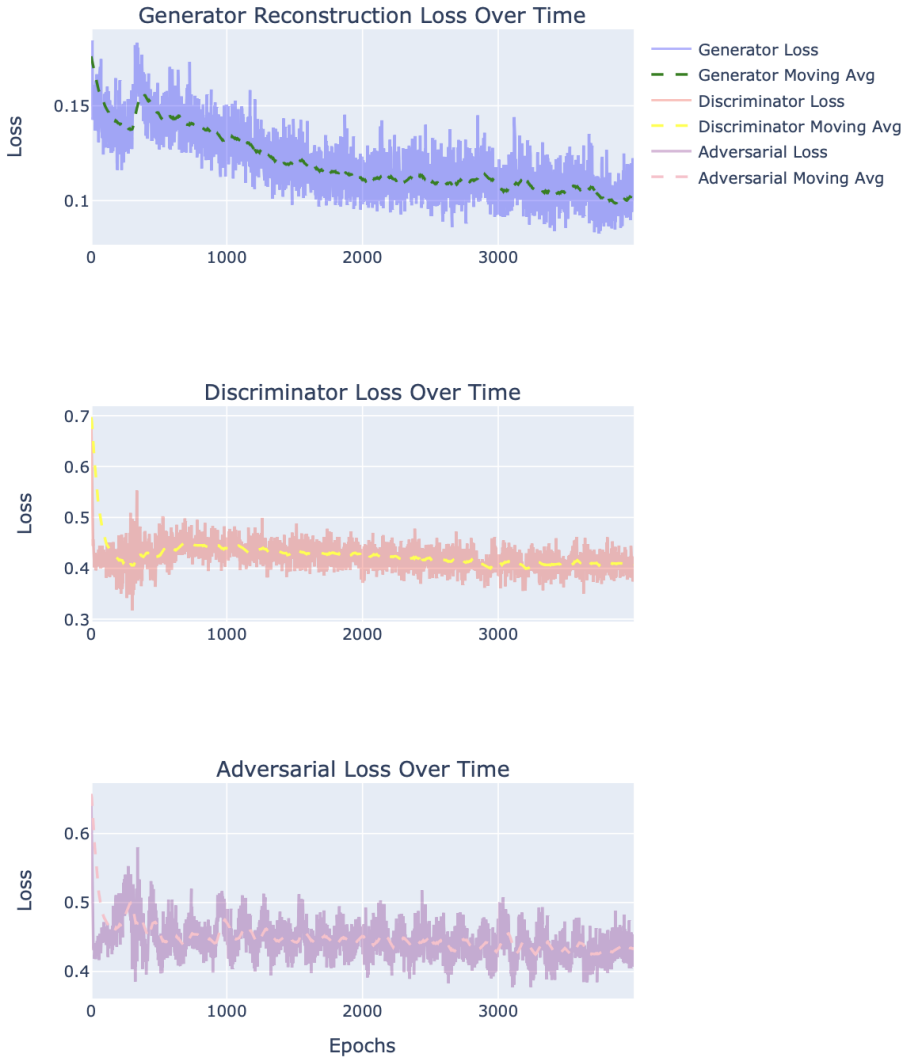

Figure 9: Loss of the model during training generator model complexity hyperparameter set to 46 and discriminator model complexity set to 4.

In 9 the discriminator model has 264341 parameters, and the generator network has 476054 parameters. Adversarial loss goes slowly down, the discriminator network cannot keep up with the generator network. As the reconstruction loss goes down, one may think that is a good thing, but in this case it means that the generator network averages on the colors which reduces resemblance and typically yields a mono-colored output with little color intensity.

This is the last example of this hyperparameter exploration.

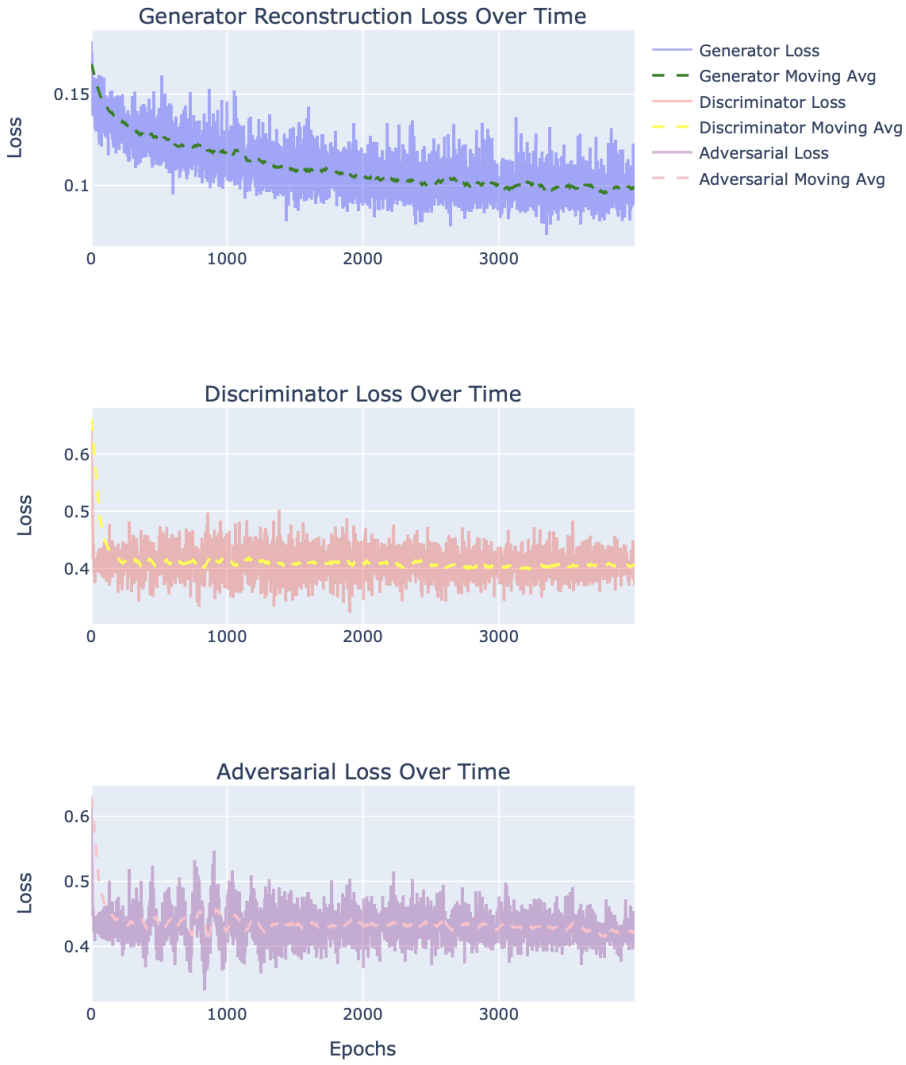

Figure 10: Loss of the model during training generator model complexity hyperparameter set to 52 and discriminator model complexity set to 4.

In 10 the discriminator model has 264341 parameters, and the generator network has 608036 parameters. Adversarial loss goes slowly down, the discriminator network cannot keep up with the generator network, and the reconstruction loss marches down. The end result of this network was not satisfying, it is displayed in the figure below.

Figure 11: Model input and output, 3x6 grid: input images in the left 3x3 grid, output images in the right 3x3 grid. Generator model complexity hyperparameter set to 52, discriminator complexity set to 4.

The output of the larger model, 52 in complexity, trained with the lower complexity discriminator model produced in the end very low intensity colors, it looks colorized, sure, but it has low intensity. Almost moving in the mono-colored direction. Is the sky really blue, or is it almost grey? For reference see 12, where the generator is smaller but the color intensity is higher. I might be going a little bit crazy after staring at so many of these collection images, I could be wrong. I am also a little color blind.

Figure 12: Model input and output, 3x6 grid: input images in the left 3x3 grid, output images in the right 3x3 grid. Generator model complexity hyperparameter set to 30, discriminator complexity set to 4.

Models could generate nice colorizations, but if they are not big enough the outputted images don't look good when inspecting them up close. For example I trained a model with generator complexity set to 42 and discriminator set to 4, one of many runs. The colorization collection is shown in 13. I would say that from afar it looks good, but up close each image looks pixelated. Look at 14 for reference. It has a very unnatural look to it. Both the trends in reconstruction loss and the adversarial loss during training looked good, but the end result was not satisfying even still. So I decided I needed to scale up the models if I wanted to achieve my goal.

Figure 13: Model input and output, 3x6 grid: input images in the left 3x3 grid, output images in the right 3x3 grid. Generator model complexity hyperparameter set to 42, discriminator complexity set to 4.

Figure 14: Model output from a generator model with complexity hyperparameter set to 38, discriminator complexity set to 4. Looking closely one can see the pixelated outlook of the image.

Conducting all of these experiment have led me to believe in a ratio of 0.8 between discriminator and generator parameters will make the reconstruction loss go down while the adversarial loss reaches a point where it is more or less stagnant. I think this condition means that the generator and the discriminator match each other during training, they both get refined to a certain point

at the same rate. Larger models tend to capture more features of the training set, and keeping the resemblance in colors high due to the adversarial tension with the model size relation of 0.8, I hope both the smoothness of the colorization improves as well as the color intensity resemblance is kept as I scale the networks up. With a larger model, I hope that the end result will look better.

5.3 Model selection

Eventually, I reach a point where I need to decide what model to be used in my colorizer program. I will here show you the final results of my image colorizer program. I ultimately decided to work with a generator complexity hyperparameter of 72 (1164456 parameters), and discriminator complexity parameter of 16 (1081937 parameters). Adversarial factor set to 0.1. I trained with a batch size of 16, learning rate of 7e-5. The end result of many colorizations, in a 3x3 grid, is displayed in figure 15. Some individual colorizations from the same model are shown in figure 5.3 and 18.

Figure 15: Model input and output, 3x6 grid: input images in the left 3x3 grid, output images in the right 3x3 grid

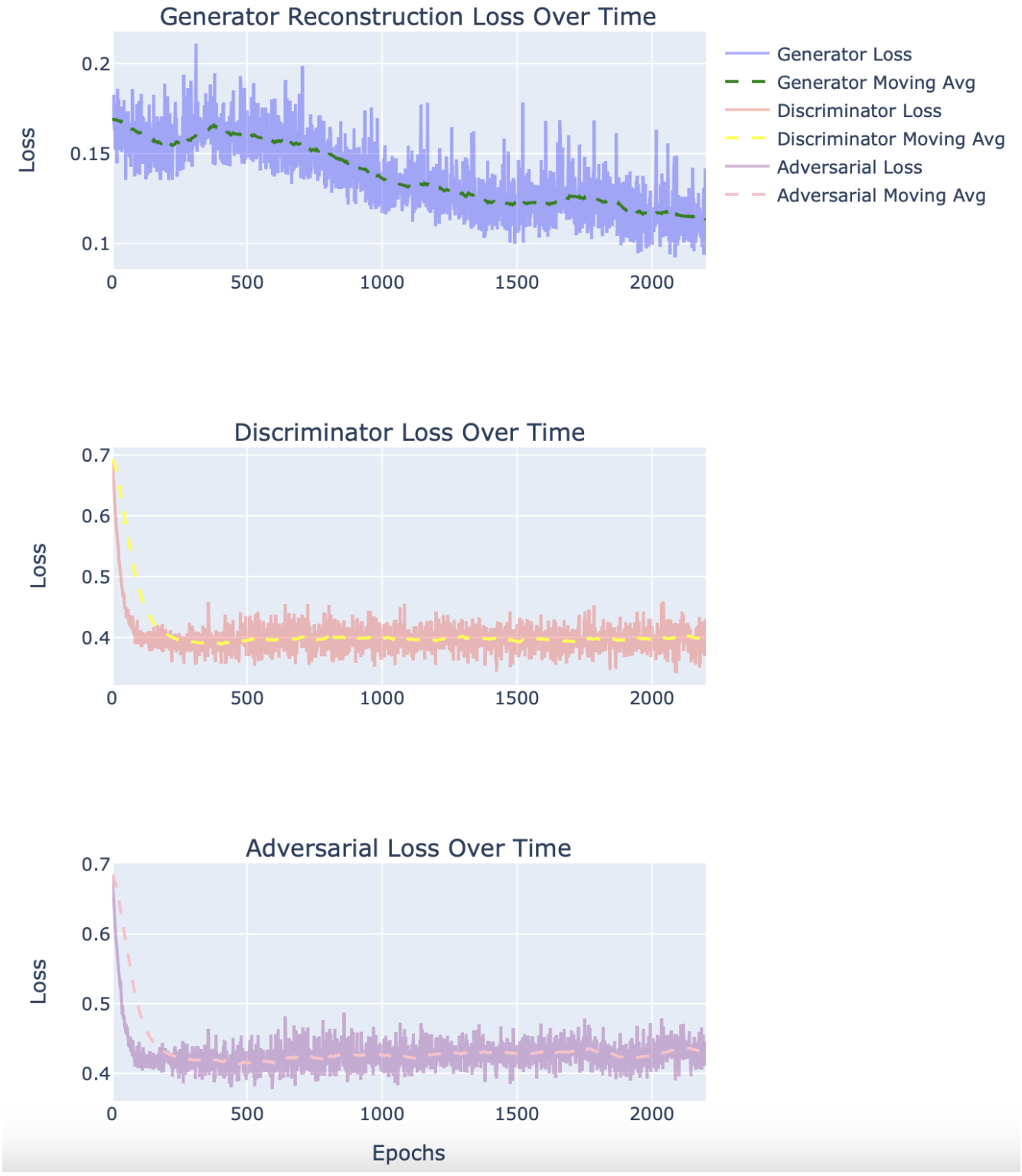

Figure 16: Loss of the selected model during training. Generator additional complexity set to 72 (476054 parameters), discriminator to 16 (1081937 parameters).

Figure 17: Model input and output: colorization of a landscape photograph featuring a very simple treehouse.

Figure 18: Model input and output: colorizing a landscape photograph featuring a hill covered in grass.

6 Lessons learned

From this project I have grown more confident about my abilities to work on deep learning projects. I will list the most notable things I've learned from this project and explain further what they meant to me.

- Importance of input processing
- Adversarial training

- Hyperparameter exploration
- Hardware evaluation

6.1 Importance of input processing

I initially started with converting original images to gray scale, then predic red, green and blue channels based on this. The results were disappointing. I tried many different hyperparameters etc. until I stumbled upon a tutorial online about this exact task. They used CIELAB and I wanted to try it out. The difference I saw at that moment, with whatever unfinished network structure I had at the moment, was staggering. Instead of predicting 3 channels of information based on 1 channel, just predicting 2 made the problem easier to overcome. After that, the metrics looked better, the end results still leaved a lot to be desired, but it was an improvement by the standard I had then. This was in an early stage of the project, where the code changed a lot and I am not sure I even traced it back then. I therefore did not save my findings, I just wanted to climb over the hill before I could write about my findings after the fact. Unfortunately, I did not pick up this flower on the way up and put it in my pocket so I could document it properly after I had achieved my goal. At that point, I grew a lot more confident that I would be able to "succeed" with the task (to the degree that I consider that I did). I had had my doubts.

6.2 Adversarial training

I experimented a lot with the network structure before I found something I believed could work. Something i struggled with was the adversarial training loop. At some point it dawned on me that the discriminator provided little to no value to the overall output of my models. At one point I abolished the discriminator completely, but then I ran into the brown, mono-colored output I have discussed in the report. I returned to the drawing board and inspected the output of the discriminator as I had implemented it. I learned that the output trended towards 0.5, which means that the discriminator is basically "giving up", and is not really interested in predicting single samples, but is more optimized to be "half-wrong half-right" all the time. I learned that the reason for that was that I had made some bad coding decisions in the training loop, where I mixed gradients in the .backward() call that shouldn't have been mixed. The error was corrected, and the code in its faulty state has not been reported. But I learned that if I did not separate between the "correct" and the "incorrect" runs, the discriminator did some kind of gradient averaging that ultimately led it to predict 0.5 all the time. The trend was that the discriminator loss would go to something like 0.69, and it took me a while to realize that that was wrong. I remain practical here, and if I am to implement an adversarial training loop again; I will surely revisit this code so I don't fall into the same trap.

6.3 Hyperparameter exploration

My intuition has been naive about deep learning. I have thought that it is "easy" one just needs "the data" and "a sufficiently large model" and the violá, project finished. This is of course not very well informed. I am glad I got the oportunity to work on this project, just to make me realize the amount of work it actually can take. Starting from nothing, it is easy to feel lost. Being pragmatic is very important. If it works, then it works. I really cant justify why my model was as deep as it was, at some point I just was satisfied with the output. I think, at some basic level,

it is hard to say exactly what set of hyperparameters will achieve ones goal. The important part is to explore the space. If the model shrinks, it may be more general, as the shrinkage acts like a form a generalization. If the model grows, it may be more precise. If the dataset is of high quality, achieving good metrics on the training set probably means performing well at new data. I am sure there are many different ways to design good models, and many ways to design bad ones. Model size seems intuitively to be a very important factor, as each node implements a decision boundary, a feature. Its expressbility grows when the model grows, but if it is overly large then it is also time and energy consuming to train. I am not sure anyone has a good foundation about of hyperparameter exploration at this point other than being practical here. But, I think if one has enough compute, this process could be completely automated.

6.4 Hardware evaluation

At Epiroc, we have access to a data platform provided by Databricks. I used one of their clusters, at one point, to test the training of my models. They charged a lot of money for this. But I wanted to test out their platform. They made it possible to write Jupyter notebooks in which I could run my project code. The result was that I became a little bit disappointed. They surely work better than on my CPU, but I learned that my macbook, using the MPS backend, had about equal performance as some of these expensive clusters. And the rates these companies charge are huge. I suspect there is a huge margin of profit for these companies at the moment. It is always best to have your own hardware, in my opinion. And you can be smarter about how you allocate the memory also. I realized I needed to implement the ImageLoader class, that only has a portion of the dataset in its memory. I hope in the future, parallel compute for these kinds of tasks will be cheaper and more available for everybody.

References

Deng, Tao (2023). "Image Colorization Based on Deep learning". In: *Degree project in Computer Science and Engineering*.