Danmarks Tekniske Universitet

02461 Introduction to Intelligent Systems

3 week project

Jonathan Simonsen (s201680)
Jacob Borregaard (s181487)
Magnus Vinjebo (s214588)

April 16, 2022

Abstract

In recent the years, the interest in automating the process of colorizing grayscale photos to produce colorful results has grown concurrently with advances in the AI field of deep learning. While highly advanced and complex models utilizing generative adversarial networks (GAN) can produce convincing results, this paper examines how the generally less complicated convolutional neural networks (CAN) complete the task. This has been done by creating our own model based on a CNN. By using photographers as human classifiers to compare the results of our own model of those of a model based on a GAN, we did not find statistical evidence to suggest that one model is better than the other. However, we found that both models are capable of colorizing photos that look realistic enough to deceive people. This finding raises troubling and highly relevant question about the trustworthiness of digital images.

Introduction - Jacob & Jonathan

Before color photography became prevalent and affordable in the 1970s, photographers usually captured their images in black and white nuances. As such, many of modern history's important events were only captured in grayscale. In many cases, this is a shame, as humans are generally more intrigued by colors than black and white. Throughout the years, this problem has been remedied using techniques such as physically hand-coloring the photos and later by using editing software such as Adobe Photoshop; however, these processes are often very time consuming as the artists need to do extensive preparatory research in order to get the colors right.

With recent years' advances in artificial intelligence, it has become increasingly more evident and advantageous to apply tools from the field of deep learning to automatically colorize black and white photos. Following this trend, the goal of this project is to create a Convolutional Neural Network (CNN) that can generate colorized versions of grayscale photos that look real to the human eye. The purpose of the project is to revitalize the past in color in a realistic manner. The study has been conducted based on the following problem statement:

How to build and train a CNN to colorize grayscale photos and how to use this technology to restore the past in color?

We are going to evaluate our results by testing how well human test subjects can identify whether a photo is original or colorized by the AI. This allows us to determine if our model creates realistic images or not. In addition, we will compare the performance of our model to that of a state of the art colorization model in the form of MyHeritage's DeOldify[1]. Based on preliminary tests, we hypothesize that our model's performance will be somewhat inconsistent and worse than that of DeOldify which will be more steady.

Methods - Jonathan & Magnus

Our CNN - Magnus

To estimate the colors of an image based on the luminosity of a grayscale picture we decided to utilize a neural network and train it to understand what the composition of an image makes it an image and how some of these components could be used to predict colors. This could have been done using pixel by pixel analysis with an ordinary non-linear neural network, but we decided to use a "Convolutional Neural Network" (CNN) which is a prominent architecture in modern computer vision. A CNN is structurally built like a non-linear neural network with the exception that all the layers have been exchanged by convolution layers, specifically engineered for image processing.
Our method entails a lot of Python libraries as many of the included features are considered boilerplate code by now. We went with PyTorch as our machine learning framework, mostly based on preference and prior experience. PyTorch supplied us with the tools to build the neural network and infrastructure for loading large datasets of images.

Before we could begin designing the architecture of the neural network, we had to set up some input/output for the script. Using a handful of other Python libraries[2], we made some internal conversions and a data loader from the public Places365 dataset[3]. Places365 is a large collection of various different image classes, including landscapes, buildings etc.. From the dataset we converted the .JPEG files to RGB-color space, and then into CIELAB-colorspace. This was done for two reasons.
Firstly, it is much easier to isolate the grayscale ground-truth from a 3-channel L*A*B* array representing the images, and secondly it can be argued that CIELAB is easier to comprehend for humans, and thus would be better to use as a target for our CNN.
We then split the images into two parts: the L* channel (the grayscale ground truth) which was used as our input, and the A*B* layers which were mapped as the output as they represent the color of the ground truth. Building the neural network required many iterations and the process was based on several different ideas

[1]Antic. 2022. DeOldify
[2]See appendix: Python Libraries
[3]Zhou. 2017. Places

of our own, however, it was influenced by supporting articles and papers. We ended up mainly taking inspiration from Luke Melas, who is a PhD student at Oxford University[4] and changed parts of his model's architecture to make it unique for us.
Specifically we changed the downscaling part from being a pretrained ResNet model to a custom cluster of layers, including Conv2D[5] and MaxPooling[6] to extract feature maps, such that our network could understand subtleties of the individual pixels and their surroundings. Due to MaxPooling, the image got reduced to approximately $\frac{1}{4}$ of the original size before we upscaled it to the original size again using PyTorch's upscaler, some more Conv2D layers, and Batch Normalization[7]. The end result was a 2-channel output where we used the loss function Mean Square Error, with regards to the A* and B* layer of the ground truth.
As a result we had the L* layer from the input and the A*B* layer from the output. To run back propagation we utilized PyTorch's built-in optimizer Adam with a learning rate of 0.001. To generate a recognizable image we then layered all 3 channels on top of each other. Because we never changed the input grayscale image, we ended up applying a sort of mask on top. This also allowed us to play around with the saturation of the output color channels, as a simple multiplier allowed us to "increase" the color saturation/contrast in post processing. Lastly we converted the CIELAB back into RGB-colorspace so that a computer monitor was able to display it properly and we would be able to evaluate the subjective performance of the network, and not just the decreasing loss-function.

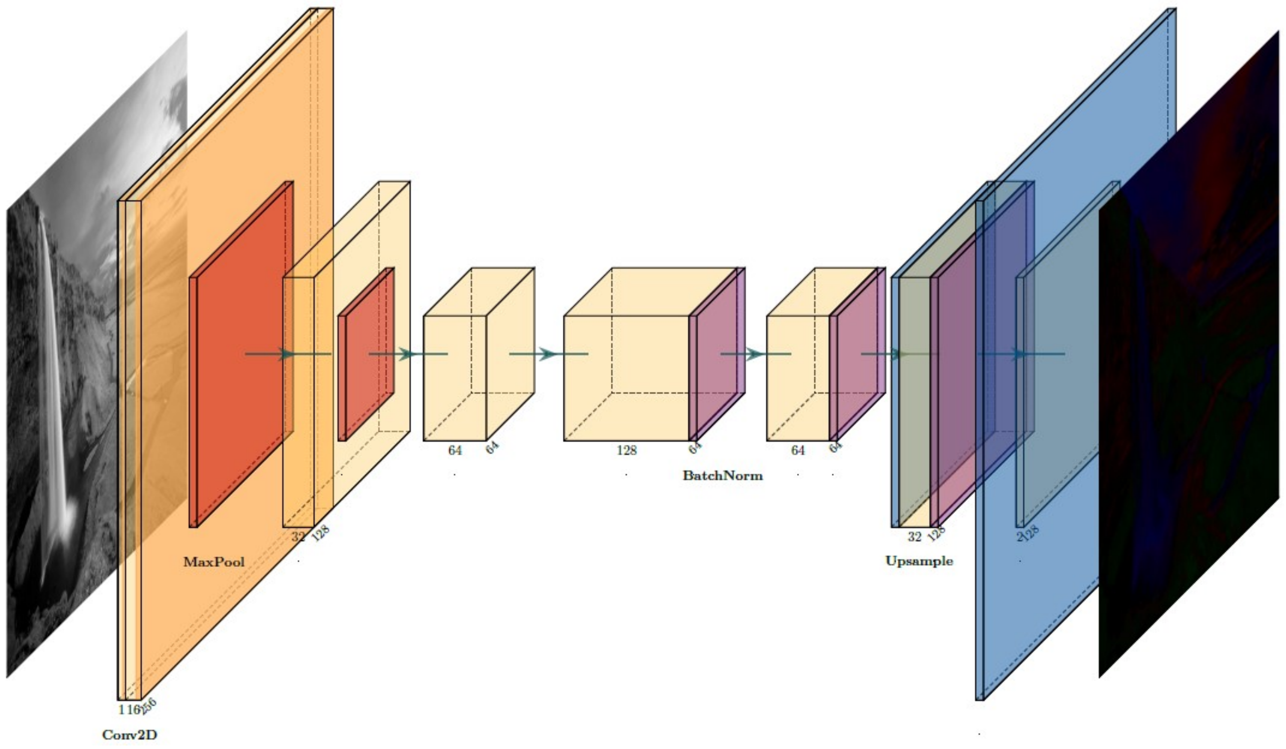

Figure 2: Architecture of our CNN

[4]Melas, Lukas. 2018: Image Colorization
[5]Stanford. 2021. CS231n course
[6]Brownlee, 2019
[7]Huber, 2020

DeOldify - A state of the art model

We compared our CNN model with the reputable DeOldify model. DeOldify has been improved several times over the years, but is currently built on a custom GAN variant, NoGAN. NoGAN has been specifically made for DeOldify by the developer Jason Antic, and is a modification that is more efficient and produces better images and videos. There has yet to be released a paper on this method, and Jason Antic mentions he isn't entirely sure what's going on internally either yet.[8] The most stable and public accessible version of the software is currently available for limited use on MyHeritage's website. Likewise this is the version we have used to create samples for our study, by supplying grayscale images for it to colorize.

Statistical considerations - Jonathan

In order to mitigate the impact of uncertainty relating to calculating our needed sample size, we decided to conduct a preliminary study. As we noticed that our model performed significantly better on images similar to those it had been trained on, we decided to only use outdoor images in the surveys. Rather than creating a single survey mixing both images colorized using our model and DeOldify, we chose to create two separate ones.
In survey A, we used our own model to colorize photos, and in survey B we used DeOldify. This allowed us to use the same photos for each model while ensuring that the participants did not see the same image colorized using both models, thus giving us the possibility to cross reference the performance of the separate models in isolated environments, without them affecting each other. Each survey had 11 questions, where the respondents in each question were shown a grayscale photo and then a colored version of the photo. The colored version could either be the original photo or a version colored using the respective model. They were then asked to pick the original version of the photo. We received 35 responses on each study and found that our model and DeOldify managed to deceive the respondents approximately 32% and 52% of the time, respectively.
Using these proportions, we calculated the sample size with a margin of error of 5 % at a 90 % confidence level and found that we needed to use 229 and 267 images in our primary surveys on respectively our model and DeOldify. We originally decided to go with a confidence level of 90 % to lower the sample size in order to save time and use the resources elsewhere, however, we ended up using a 95 % confidence level after giving the situation more thought. The primary study followed the same design as the preliminary study, however, this time we decided to ask two photographers to participate rather than collecting responses from multiple random people. The first photographer completed a survey where images colorized using our model had been used, the other one completed a survey where the images used had been colorizing using DeOldify. However, while we decided to use 260 images in survey A, we could only use 16 images in survey B as we reached the limit on MyHeritage. Ideally, we should have used the same amount of images and the same images in each survey.

[8] Antic, 2021: What is NoGan

Results - Jacob

The confusion matrix in figure 3 shows the results of our test with our own CNN in percentage and actual numbers. Our test results show that our test subject was able to correctly guess the original photos 100% of the time, while he only guessed 73% of the generated pictures correctly. This means that our model was able to deceive the subject on 27% of the images.

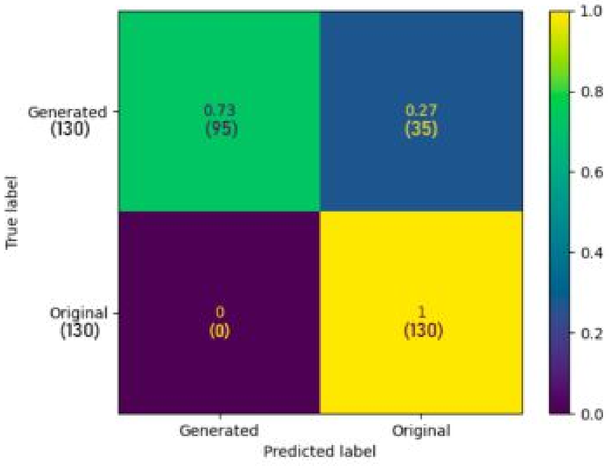

Figure 3: CNN - Confusion Matrix

Table 1: Confidence Interval: 95% Confidence Level

Model	Success Rate	Confidence Interval
CNN	26.9%	19.5% - 35.4%
Deoldify	50.0%	15.7% - 84.3%

Figure 4 shows the results of the test with the DeOldify. As shown in the matrix, this test was done with far less images, but the results show that the model was able to deceive our test subject 50 % of the time.

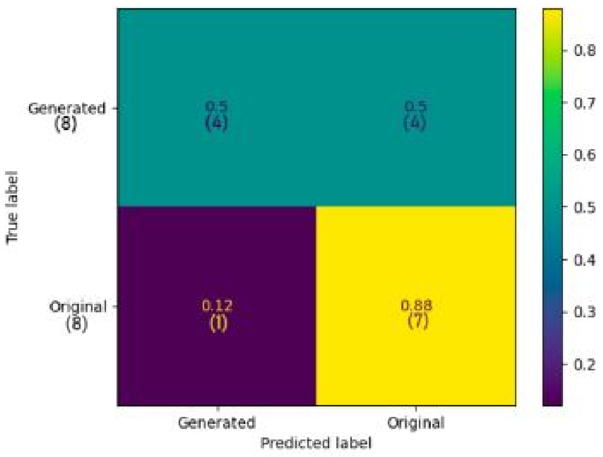

Figure 4: DeOldify - Confusion Matrix

Table 1 illustrates a comparison between the results of the two models. The DeOldify model averagely performed better than our model. However, the small sample size of the test means that the confidence interval is very large, as the intervals of respectively 19.5% - 35.4% and 15.7% - 84.3% overlap and thus the difference in performance is not significant enough to conclude anything when comparing the performance of the two models.

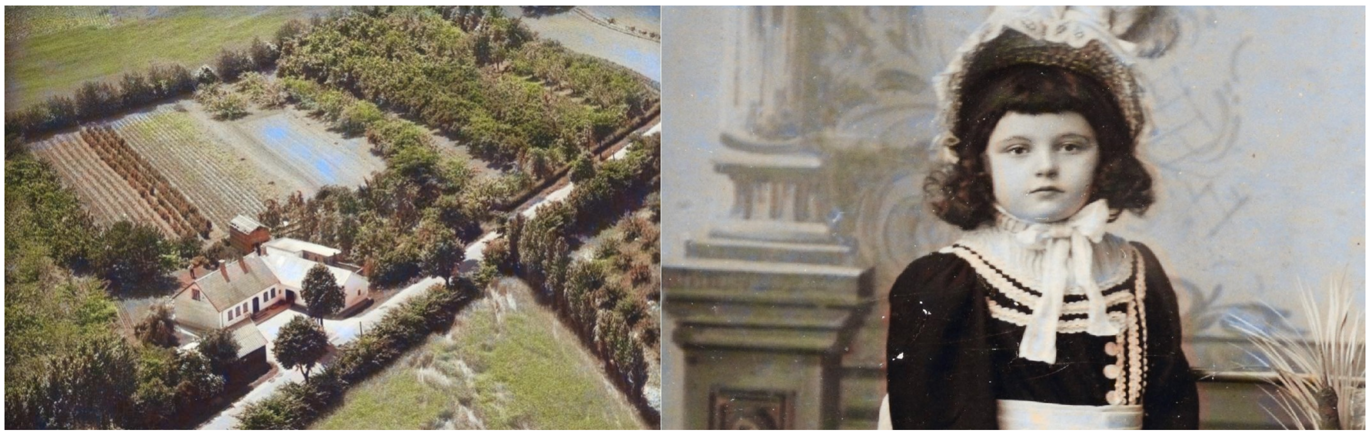

Figure 5: Examples of old images colorized with our model

Discussion - Jonathan & Magnus

In order for the results from the study to have been more comparable, we should ideally have used the same sample size for the study on DeOldify. This would have made the range of the respective confidence interval considerably smaller which then potentially could have resulted in non-overlapping confidence intervals. In relation to this, it is important to note that while we based on the preliminary study calculated our sample size to be approximately 260, the actual sample size used in the study on our model was only 130. This was due to the fact that we used 130 original images and 130 images which had been colorized with the model. The same proportionate reduction in sample size was also made for the study on DeOldify. Ideally, we should have used 260 images colorized using the model as dictated by the calculation, but we did not come to this realization until after we conducted the study.

Our findings suggest that it is possible to use even a relatively simple deep learning model to generate colorized images that look decently authentic. As such, these models can potentially give the average individual a chance to see for example old family photos from generations back in color. Looking at the bigger picture, these technologies present humanity with an easily accessible approach to explore our cultural heritage in a different way. Aside from this, the concept of using AI to colorize grayscale input can also be utilized in other areas, particularly in computer vision.

From studying and developing our own CNN, we made some discoveries regarding the limitations of image processing, which also applies to the field in general. Firstly, we saw a common tendency in several models to perform advantageously on images similar to those of the training data-set. As an example our model performed poorly on images of humans when trained on images of places and vice versa. This heavy bias towards the training data potentially results in color that poorly represents the ground truth of the image. Secondly, we saw many occurrences of what researchers at Stanford have coined "the averaging problem", which is a result of the fact that colorization is hard for computers because it is ill-posed, meaning that there are multiple color images which are equally "correct" given a grayscale root version. This means the algorithm either has to make a bold choice and guess which color is the correct one, or make a compromise, averaging the possibilities, which leads to more brownish/beige color nuances.

During our evaluation of the performance of our CNN, we noticed that often when it failed to produce a satisfactory result it came down to one or more of three primary errors in addition to the averaging problem. Firstly, it had a tendency to use too much blue when generating the output, resulting in blue spots on the image, as it also can be seen on the images in figure 5. Secondly, it also appeared to have this issue with red spots. Thirdly, the model often had problems with shadows likely due to the change in luminosity around areas with shadows. We were also told by the photographer who evaluated the images colorized by DeOldify that this model also had a tendency to struggle with the latter.

While convolutional neural networks and other deep learning models can produce fascinating results when it comes to colorizing grayscale images that can give us a different perspective on history, we must not neglect to consider the ethical aspects relating to this. As mentioned, many colorization models have a tendency to use less vibrant colors and therefore they often output images that make the settings appear more dull and colorless than they actually were when the photo was taken[9]. This unintentionally gives the impression that the past was pale and lifeless, which certainly was not the case. To put it in another way, artificial intelligence does not understand human culture and history the same way humans do, and thus, historical and cultural accuracy continues to be a great challenge for these deep learning models[10]. Building on this, it can be argued that colorization of old grayscale images should be left to human experts, who through research and knowledge about the time period and context in question can pick more appropriate colors and thereby create a historically speaking more accurate representation. This also luminates another critical issue; colorization of grayscale photos is by definition image manipulation and as such, the approach, alongside other image generation technologies like deep-fakes and NVIDIA Canvas, raises troublesome and highly relevant questions about the trustworthiness of digital images.

[9] Goore 2021
[10] McCarty 2021

References

Antic, Jason. November 3rd 2021. "DeOldify".
Accessed April 16, 2022:
https://github.com/jantic/DeOldify

Melas, Lukas. May 15th , 2018. "Image Colorization with Convolutional Neural Networks".
Accessed April 16, 2022:
https://lukemelas.github.io/image-colorization.html

Stanford, Vision. Spring 2021. "Convolutional Neural Networks (CNNs / ConvNets)".
Accessed April 16, 2022:
https://cs231n.github.io/convolutional-networks/

Brownlee, Jason. July 5th 2019. "Machine Learning Mastery: A Gentle Introduction to Pooling Layers for Convolutional Neural Networks".
Accessed April 16, 2022:
https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks

Huber, Johann. November 6th, 2020. "Batch normalization in 3 levels of understanding"
Accessed April 16, 2022:
https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338

Goree, Sam. April 4, 2021. "The Limits of AI Image Colorization: A Companion". Github.
Accessed April 16, 2022
https://samgoree.github.io/2021/04/21/colorization_companion.html

McCarty, Niko. 2021. "AI can't color old photos accurately. Here's why." Scienceline, January 13, 2021. https://scienceline.org/2021/01/ai-cant-color-old-photos/

Zhou, Bolei and Lapedriza, Agata and Khosla, Aditya and Oliva, Aude and Torralba, Antonio. 2017.IEEE Transactions on Pattern Analysis and Machine Intelligence: "Places: A 10 million Image Database for Scene Recognition". http://places2.csail.mit.edu/index.html

Learning outcome

During this project, we spent a lot of time learning about different models within the field of deep learning, with a particular focus on convolutional neural networks and generative adversarial networks. We also acquired knowledge about the python library PyTorch as we used this extensively to design our own model. This process was time-consuming, as we spent a lot of time trying to optimize our model without experiencing any significant improvements - we could have used these resources elsewhere. One of our main takeaways from this project is to make sure to establish the experiment design early on in the process. We ended up changing our experiment two days before hand-in, which resulted in us ending up with a half-baked experiment. This meant that it was hard for us to conclude anything based on our results.

Appendix

Python Libraries

- PyTorch
- MatPlotLib
- Numpy
- Pillow
- Skimage

Loss function - plot

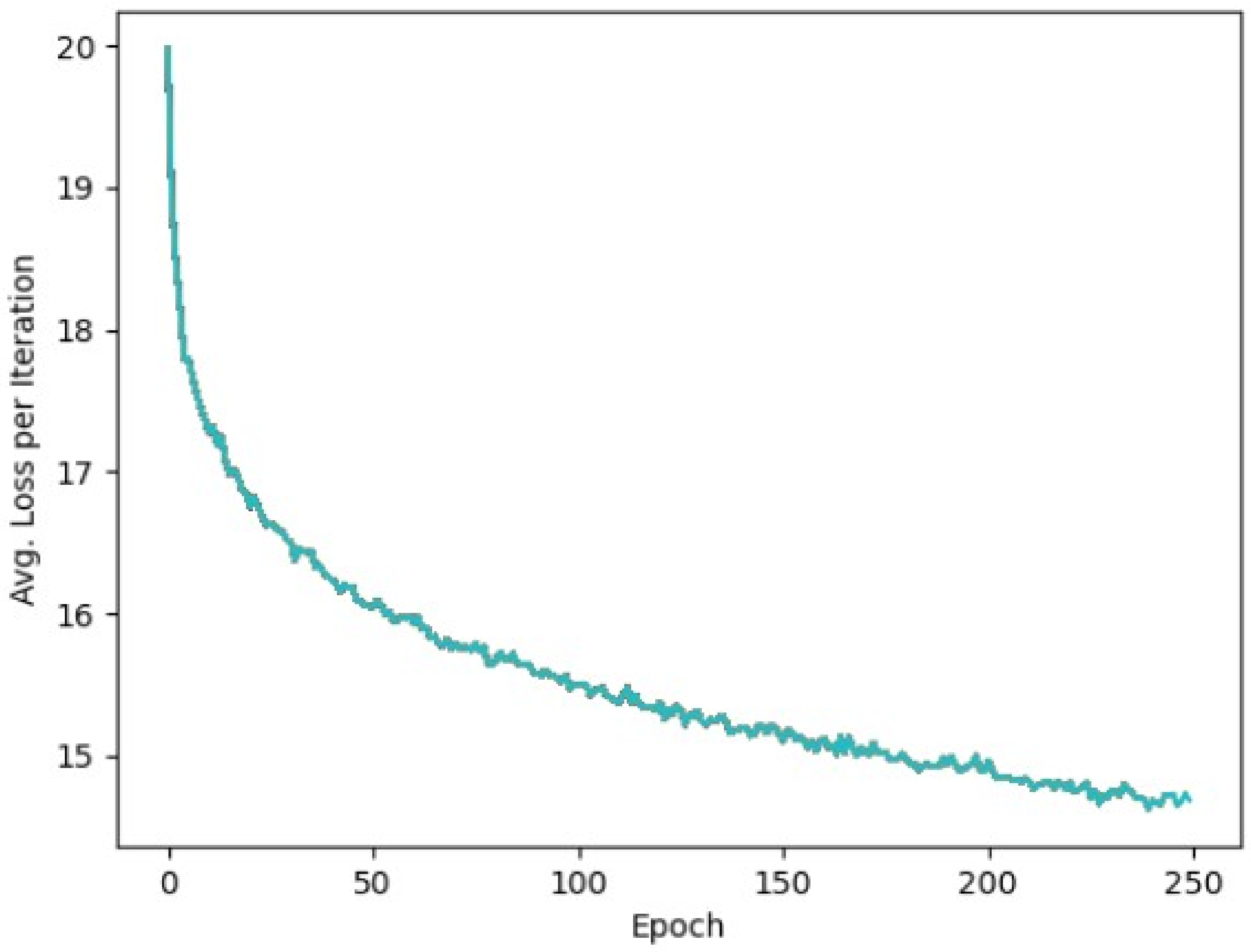

Figure 6: Loss function

Results - preliminary study

Table 2: Each model's performance on the individual pictures

Model	1	2	3	4	5	6	7
CNN	18.5%	29.6%	29.6%	29.6%	37%	40.7%	37%
Deoldify	44.4%	47.2%	69.4%	47.2%	62.9%	62.9%	30.6%

Table 3: Confidence Interval: 95% Confidence Level

Model	Mean Success Rate	Confidence Interval
CNN	31.7%	27.1% - 36.3%
Deoldify	52.2%	43.7% - 60.7%

Main training script and Image colorizer script

Below you will find the full code for both the training script of the CNN model, and the code for the generator used to colorizer multiple images in row. Both of the scripts and a lot more supporting structure can be found at the following github:
https://github.com/Magdk01/Image-Colorization-using-basic-CNN

Training_script.py

```python
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
from PIL import Image
from skimage.color import rgb2lab, lab2rgb
import os, time
from torch.utils.data import Dataset

# These are the main variables to change if you don't want to poke around in the
code.

# Batch_size is the amount of pictures per batch thats parsed through the model
batch_size = 4
# Epochs are the amount of times to go through the entire dataset
epochs = 1
# Learning_rate is the rate of which the optimizer is allowed to change the
weights and biases
learning_rate = 0.001

# PATH should be changed for each training as to not overwrite previous
tranings.
# Todo make automatic naming shceme
PATH = './Trained_Models/Main_Gen1.1_Model_1Epoch_hackeddataset_MSVN.pth'

# PATH is also the PATH that has to be used in generator to test the model on
more images


# Composed transform layer for image augmentation
transform = transforms.Compose(
    [transforms.RandomHorizontalFlip(),
     transforms.ToTensor(), ])

# Defining system paths to  locate images for custom dataset
image_path = './data/val_256_new'
train_image_paths = []
for pictures in os.listdir(image_path):
    train_image_paths.append(os.path.join(image_path, pictures))

# Supporting variables to store classes matching images
classes = []  # to store class values
idx_to_class = {i: j for i, j in enumerate(classes)}
class_to_idx = {value: key for key, value in idx_to_class.items()}


# Defining the custom dataset based on torchvision.Dataset Class so images can
be dataloaded
class CustomDataset123(Dataset):
    def __init__(self, image_paths, transform=False):
        self.image_paths = image_paths
        self.transform = transform
    def __len__(self):
        return len(self.image_paths)

    # getitem needs overhaul to allow for pickling, so it enables
multiproccesing for dataloader
    def __getitem__(self, idx):
        image_filepath = self.image_paths[idx]
        image = Image.open(image_filepath)
        image = image.convert('RGB')
        label = image_filepath.split('/')[-2]
        label = list(class_to_idx[label])

        if self.transform is not None:
            image = self.transform(image)

        return image, label


# Both of the training sets. CustomDataset123 contains 4.700 images, but
requires aditional setup to run.
# Places365 is the generic dataset provided directly from torchvision and can be
downloaded in-code.

# trainset = CustomDataset123(image_paths=train_image_paths,
transform=transform)
trainset = torchvision.datasets.Places365(root='./data', split='val',
small=True, download=False,
                                          transform=transform)

# The loader works for both datasets, however num_workers has to be set to 0,
for CustomDataset123,
# until pickling has been fixed
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
shuffle=True, num_workers=2)

if __name__ == '__main__':

    # If a cuda device is avaliable, the traning will prioritize this device
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print(f"Using {device} device")

    # Starting a timer to monitor training time
    start_time = time.time()

    # Before training starts, a batch of pictures matching the batch size is
stored to view the subjective succes
    # of the image processing
    dataiter = (iter(trainloader))
    images, labels = dataiter.next()

    # Lots of IO to fix the transformations from arrays to tensors and Vice
Versa
    original_set_tensor = images
    original_set_numpy = original_set_tensor.numpy()
    original_set_numpy = np.transpose(original_set_numpy, (2, 3, 0, 1))
    original_set_lab = rgb2lab(original_set_numpy)
    copy_of_lab = np.copy(original_set_lab).astype('float32')
    copy_of_lab = np.transpose(copy_of_lab, (2, 3, 0, 1))
    grayscale_copy_of_lab = copy_of_lab[:, 0]
    grayscale_copy_of_lab =
torch.from_numpy(grayscale_copy_of_lab.astype('float32'))
    grayscale_copy_of_lab = torch.unsqueeze(grayscale_copy_of_lab, 1)


    # Generator is the class representing the architecture of the CNN. Is a
child of PyTorch's nn.Module
    class Generator(nn.Module):

        def __init__(self):
            super().__init__()
            self.model = nn.Sequential(

                # Downsample
                nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1),
                nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
                nn.MaxPool2d(kernel_size=2),
                nn.ReLU(),
                nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
                nn.MaxPool2d(kernel_size=2),
                nn.ReLU(),
                nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),

                # Upsample
                nn.Conv2d(128, 64, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(),
                nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(),
                nn.Upsample(scale_factor=2),
                nn.Conv2d(64, 32, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(),
                nn.Conv2d(32, 2, kernel_size=3, stride=1, padding=1),
                nn.Upsample(scale_factor=2)
            )

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

    # Instantiation of the generator
    gen = Generator().to(device)

    # Setting up both the criterion and optimizer such that they can be modfied.
For this MSELoss and
    # PyTorch's Adam optimizer has been utilized
    criterion = nn.MSELoss()
    optimizer = torch.optim.Adam(gen.parameters(), lr=learning_rate)

    # Some variables to store loss
    total_loss = 0
    epoch_loss_list = []

    # Main Training loop. Epochs are based on value given to the variable
    for epoch in range(epochs):
        epoch_loss = 0
        for i, data in enumerate(trainloader):

            # Data is loaded from the dataloader, and labels are store in a
variable with is just a placeholder.
            inp_img_inside, labels = data

            # Images goes through transformation from Tensor to numpy array
            original_set_tensor_inside = inp_img_inside
            original_set_numpy_inside = original_set_tensor_inside.numpy()
            original_set_numpy_inside = np.transpose(original_set_numpy_inside,
(2, 3, 0, 1))
            # Then converting images from RGB to CIELAB
            original_set_lab_inside = rgb2lab(original_set_numpy_inside)
            # Then transposed back to tensor dimensions
            new_set_lab_inside = np.transpose(original_set_lab_inside, (2, 3, 0,
1))
            # A copy is then made of the full CIELAB spectrum, such that the L*
channel can be used for input and
            # the A*B* channels can be used for MSELoss
            lab_grayscale_inside = np.copy(new_set_lab_inside)
            # L* channel
            lab_grayscale_inside = lab_grayscale_inside[:, 0]
            lab_AB_inside = np.copy(new_set_lab_inside)
            # A*B* Channels
            lab_AB_inside = lab_AB_inside[:, [1, 2]]

            lab_grayscale_inside =
torch.from_numpy(lab_grayscale_inside.astype('float32'))
            # Unsqueezing L* channel to mimic 1 dimensional image channel
            lab_grayscale_inside = torch.unsqueeze(lab_grayscale_inside, 1)
            lab_target_inside =
torch.from_numpy(lab_AB_inside.astype('float32'))

            # Resetting the optimizers weights
            optimizer.zero_grad()

            # Forwarding grayscale image trough the CNN
            output = gen(lab_grayscale_inside.to(device))

            # Calculating loss over output and the ground truth A*B* channels
            loss = criterion(output, lab_target_inside.to(device))

            # Backpropagation according to loss
            loss.backward()

            # Stepping the optimizer for improving model
            optimizer.step()

            # Variables that accumulate loss
            total_loss += loss.item()
            epoch_loss += loss.item()

            # Print loop to track progress of the training
            if i % 250 == 0:
                print(f'Epoch = {epoch}, I = {i},  Loss: {total_loss / 250},
Time: {time.time() - start_time}')
                total_loss = 0

        # Below is the code for plotting the training progress over epochs.
        epoch_loss_list.append(epoch_loss / 4700)
        plt.plot(epoch_loss_list)
        plt.xlabel('Epoch')
        plt.ylabel('Avg. Loss per Iteration')
        # If your python doesn't allow the script to continue before closing the
figure windows.
        # (if you don't have SciView etc.)
        # Comment "plt.show()" out, so that it doesn't pause the training after
each epoch
        plt.show()

    # After the training is done, the weights and bias' at the defined path
    torch.save(gen.state_dict(), PATH)

    # Image transformations to allow it to be displayed to monitor models
performance
    GeneratedAB_img = gen(grayscale_copy_of_lab.to(device))
    GeneratedAB_img = GeneratedAB_img.cpu()
    GeneratedAB_img = GeneratedAB_img.detach().numpy()
    # concatenating the L* input channel with the A*B* output channels to create
a 3-channel RGB
    merged_img = np.concatenate((grayscale_copy_of_lab, GeneratedAB_img),
axis=1)

    fig = plt.figure(figsize=(10, 7))
    # Displays the groundtruth images
    for index in range(batch_size):
        fig.add_subplot(3, batch_size, index + 1)
        plt.axis('off')
        plt.imshow(original_set_numpy[:, :, index])
    # Displays the grayscale of the ground truth images
    for index in range(batch_size):
        fig.add_subplot(3, batch_size, batch_size + index + 1)
        plt.axis('off')
        plt.imshow(original_set_numpy[:, :, index, 0], cmap='gray')
    # Displays the generated colorization
    for index in range(batch_size):
        fig.add_subplot(3, batch_size, batch_size * 2 + index + 1)
        plt.axis('off')
        img_slice = merged_img[index]
        print_img = np.transpose(img_slice, (1, 2, 0))
        plt.imshow(lab2rgb(print_img))
    plt.show()
```

Image_colorizer.py

```
import os.path
import torch
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import torch.nn as nn
from skimage.io import imread
from skimage.color import rgb2lab, lab2rgb

# Saved weights for the CNN model - Can be any of the trained weights matching
Main generation 1.1.
# If other generations are required the Generator Class will need to be updated
as well
PATH = './Trained_Models/Main_Gen1.1_Model_10Epochs_places365_valset_JB.pth'

# Color saturation multiplier
ColorSat_Multiplier = 2

# To use the generator, simply put .JPG images either in color or grayscale
# (Doesn't really matter as the grayscale layer is isolated anyway)
# into a folder in the same directory called "tobeGen" and supply a folder
called "generated" in the same location.
# Then this program will loop through all images in "tobeGen" folder and put
them in "generated" folder

if __name__ == '__main__':

    # If a cuda device is available, the training will prioritize this device
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print(f"Using {device} device")


    # Generator is the class representing the architecture of the CNN. Is a
child of PyTorch's nn.Module.
    # This generator is generation 1.1 and can as such only support Gen1.1
weights

    class Generator(nn.Module):

        def __init__(self):
            super().__init__()
            self.model = nn.Sequential(

                # Downsample
                nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1),
                nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
                nn.MaxPool2d(kernel_size=2),
                nn.ReLU(),
                nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
                nn.MaxPool2d(kernel_size=2),
                nn.ReLU(),
                nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
                nn.ReLU(),
                # Upsample
                nn.Conv2d(128, 64, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(),
                nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(64),
                nn.ReLU(),
                nn.Upsample(scale_factor=2),
                nn.Conv2d(64, 32, kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(32),
                nn.ReLU(),
                nn.Conv2d(32, 2, kernel_size=3, stride=1, padding=1),
                nn.Upsample(scale_factor=2)
            )

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


    # Instantiation of the generator and loading the trained wieghts
    net = Generator().to(device)
    net.load_state_dict(torch.load(PATH))


    # Helper function for loading images and resizing them to be divisible by 8
- #todo fix clunky code
    def load_image(RGB_image_filename):
        RGB_array = imread(os.path.join('./tobeGen', RGB_image_filename))
        LAB_array = rgb2lab(RGB_array).astype('float32')
        width = np.size(RGB_array[:, 0, 0])
        height = np.size(RGB_array[0, :, 0])

        idx = 0
        idx2 = 0
        while (width / 2 / 2 / 2) % 1 != 0:
            width += 1
            idx += 1

            print(f'width: {width}, idx = {idx}')
            if idx >= 40:
                break
        while (height / 2 / 2 / 2) % 1 != 0:
            height += 1
            idx2 += 1

            print(f'height: {height}, idx = {idx2}')
            if idx2 >= 40:
                break

        img_size = width, height

        return RGB_array, LAB_array, img_size

    # Helper function to seperate L*-channel and A*B*-Channel.
    # Then forwarding through the model and contecating the layers to create a
displayable RGB image
    # For more information either read the belonging raport of the commens of
the main.py of image processing
    def process_image(LAB_array, img_size):
        width, height = img_size
        L_array = LAB_array[..., 0]
        L_tensor = transforms.ToTensor()(L_array)
        L_tensor = transforms.Resize((width, height))(L_tensor)
        L_tensor = torch.unsqueeze(L_tensor, 0)
        L_array = L_tensor.numpy()
        Generated_AB_tensor = net.forward(L_tensor.to(device=device))

        Generated_AB_array = Generated_AB_tensor.cpu().detach().numpy() *
ColorSat_Multiplier
        Merged_LAB_array = np.concatenate((L_array, Generated_AB_array), axis=1)
        Merged_LAB_array = Merged_LAB_array[0]
        Transposed_LAB_array = np.transpose(Merged_LAB_array, (1, 2, 0))
        Merged_RGB_array = lab2rgb(Transposed_LAB_array)

        return Merged_RGB_array

    # Simple dipslays helper function to see comparisons of original image and
the generated image
    def display_image_pair(RGB_array, new_RGB_array):
        fig = plt.figure(figsize=(10, 7))
        fig.add_subplot(1, 2, 1)
        plt.imshow(RGB_array)
        fig.add_subplot(1, 2, 2)
        plt.imshow(new_RGB_array)
        plt.show()

    # Img_save takes all of the images stored in folder "tobeGen"
    # and puts them through the model and stores them in folder "generated"

    def img_save(new_RGB_array, inp_filename):
        main_path = './generated'
        save_path = os.path.join(main_path, inp_filename)
        print(save_path)
        try:
            plt.imsave(save_path, new_RGB_array)
        except ValueError:
            print(f'{inp_filename} already has a generated picture in the
folder')

    # Loop that goes through all the files in folder "tobeGen" and applies all
the helper functions
    for filename in os.listdir('./tobeGen'):
        RGB_array, LAB_array, img_size = load_image(filename)

        new_RGB_array = process_image(LAB_array, img_size)

        display_image_pair(RGB_array, new_RGB_array)

        img_save(new_RGB_array, filename)
```