Image Colorization

Serghei Socolovshi (serghei@kth.se) and Angel Igareta (alih2@kth.se)

Scalable Machine Learning and Deep Learning

Project Report

Stockholm, December 2020

1 Introduction

Before Artificial Intelligence, image colorization was reserved to artists that aimed to give the original colors to a picture. In fact, nowadays, most image colorization tasks are done manually by using image editing software such as Photoshop by professional studios. Despite it is a challenging problem due to the multiple image conditions that need to be considered, deep learning techniques have recently achieved promising results in this field [5].

This project aims to study the Image Colorization problem of the Computer Vision branch and implement a Deep Neural Network able to colorize black and white images.

2 Tools

The tools used to develop this project are:

- Python 3.7.
- Tensorflow 2.4 with Keras: open-source framework for machine learning and deep learning applications. It was used to preprocess input data, create data pipelines and train models with different Keras layers.
- Datalore: cloud service created by JetBrains that facilitates collaborative work with Jupyter Notebook.
- Various python libraries, such as NumPy and matplotlib.

3 Data

Regarding the data, two public datasets were used:

- Flickr 30k dataset: a public dataset which contains 30 thousand images in 200x200 resolution. It has become a standard benchmark for image captioning, but because of the variety of images that it contains, this dataset was used for final training and validation of the models [2].
- Flickr 8k dataset: a subset of the previous dataset containing 8 thousand images in 200x200 resolution. It was used mostly for initial training of the models to understand the performance of the implemented models [3].

4 Data Processing

In order to train the models, the datasets described above had to be preprocessed before being provided to the models. The original color space of the images was the infamous RGB representation, which represents the color from an image using three different channels: Red, Green and Blue. However, this color space does not approximate to the human vision, and previous literature has shown that better results can be achieved by using the LAB color space [1].

LAB format splits the image in 3 channels:

- L: Lightness of the image on a scale from 0 to 100. This channel represents a grayscale image.
- a: green-red color spectrum. The values range from -128 (green) to 127 (red).
- b: blue-yellow spectrum. The values range from -128 (blue) to 127 (yellow).

In the preprocessing, the L channel needs to be separated from the rest, as it will be used as the input for the model, while the combination of a*b channels will be used as ground truth for the prediction. Once all the images have been normalized, converted to LAB format and their channels separated, the dataset is divided into training and validation sets using 80:20 split. All these tasks were implemented using tf.data.Dataset API [4], which supports writing efficient pipelines for the input data.

5 Models

The models were trained using the L channel of the images as the input, trying to predict the a*b channels of the image. The trained architectures were the following:

5.1 CNN Model

The first model trained was inspired by the architecture used in the Colorful Image Colorization paper [8]. The model is build of 8 blocks of convolutional layers, without any pooling layers and using only batch normalization at the end of each block, as the only regularization. The implemented model did not follow exactly the proposed architecture: it does not have a softmax activation layer towards the end of the network and it uses standard Adam optimizer and Mean Square Error loss.

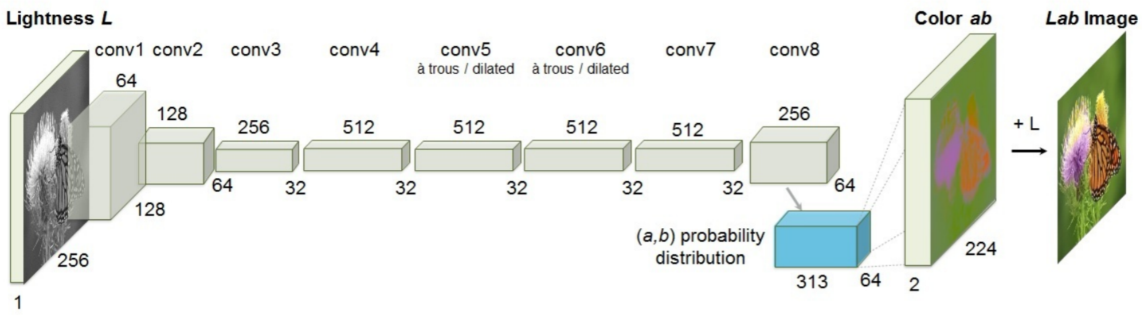
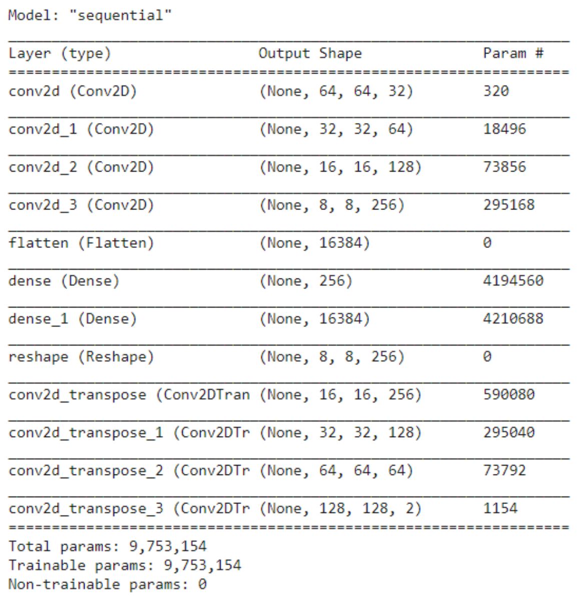


Figure 1: CNN Model Architecture [8]

5.2 Autoencoder Model

Te following architecture was adapted from a blog on TowardsDataScience website, where the author was testing different deep learning models for the image colorization using screenshots from Wario Land games [7]. One of the proposed models was an autoencoder that consisted of an encoder part with 4 convolutional layers; the central block of layers with one Flatten, two Dense and one Reshape layers; and the decoder part with 4 convolutional transpose layers. The training was done using RSMprop optimizer and Mean Square Error loss.

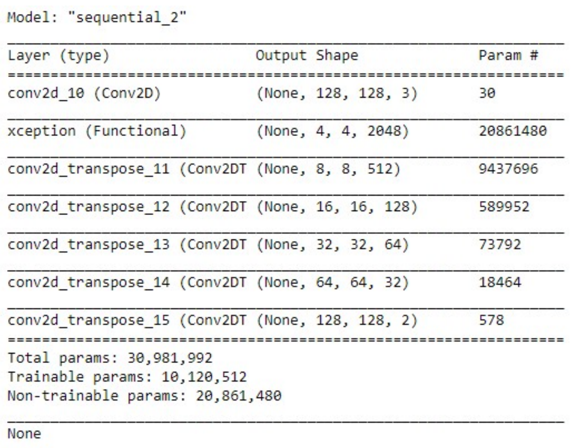


```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 64, 64, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 64)        18496
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 128)       73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 256)         295168
_________________________________________________________________
flatten (Flatten)            (None, 16384)             0
_________________________________________________________________
dense (Dense)                (None, 256)               4194560
_________________________________________________________________
dense_1 (Dense)              (None, 16384)             4210688
_________________________________________________________________
reshape (Reshape)            (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 16, 16, 256)       590080
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 32, 32, 128)       295040
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 64, 64, 64)        73792
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 128, 128, 2)       1154
=================================================================
Total params: 9,753,154
Trainable params: 9,753,154
Non-trainable params: 0
```

Figure 2: Autoencoder Model Architecture [7]

5.3 CNN using Pretrained Model

The last implemented architecture was a combination of Xception pretrained model [6] that acted as an encoder for the input image and five convolutional layers that decode this representation to the expected output. During the training stage, the Adam optimizer and Mean Square Error loss were used.



```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_10 (Conv2D)           (None, 128, 128, 3)       30
_________________________________________________________________
xception (Functional)        (None, 4, 4, 2048)        20861480
_________________________________________________________________
conv2d_transpose_11 (Conv2DT (None, 8, 8, 512)         9437696
_________________________________________________________________
conv2d_transpose_12 (Conv2DT (None, 16, 16, 128)       589952
_________________________________________________________________
conv2d_transpose_13 (Conv2DT (None, 32, 32, 64)        73792
_________________________________________________________________
conv2d_transpose_14 (Conv2DT (None, 64, 64, 32)        18464
_________________________________________________________________
conv2d_transpose_15 (Conv2DT (None, 128, 128, 2)       578
=================================================================
Total params: 30,981,992
Trainable params: 10,120,512
Non-trainable params: 20,861,480
_________________________________________________________________
None
```

Figure 3: Combination Model Architecture

6 Implementation

The code was implemented using Jupyter Notebook, so it can be executed using any environment that allows this file format. It can be divided in the following main sections:

- Data Preprocessing: This section covers the preprocessing applied to the input data, including the conversion from rgb image to lab, the creation of the training and validation sets, and the configuration of tensorflow data pipelines for a higher performance during training.
- Models: It contains the implementation using Keras of the three models presented in the previous section, alongside the trainable parameters for each one.
- Train Stage: One of the previous models can be selected and trained with the images found in the training set. In each epoch, the validation accuracy and loss is printed. At the end of the training, the accuracy-loss plot is displayed for both train and validation set.
- Test Stage: It retrieves one batch of the validation set and prints 4 predictions compared to their original images.
- Convert custom pictures: An image path can be added containing a black and white image and the best model is used to predict its respective image in color.

7 Results

The results obtained per model are:

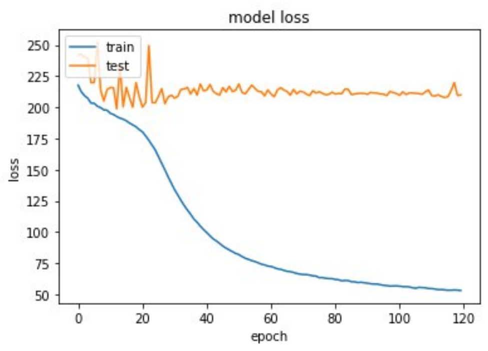
7.1 CNN Model

The model consisting of a Convolutional neural network obtained the following results in regards of the accuracy and loss both in training and validation stages after 120 epochs.

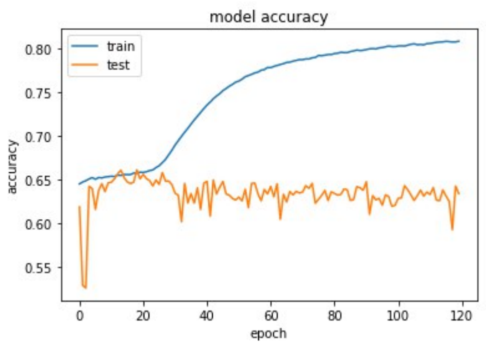
- Train Loss: 53.5244 MSE
- Train Accuracy: 80.80%
- Validation Loss: 209.4783 MSE

- Validation Accuracy: 64.27%

Also, the training and validation accuracy-loss plots can be observed in the figure **??**. It can be appreciated a severe overfitting from epoch 30.



(a) CNN Model - Loss chart (120 epochs)



(b) CNN Model - Accuracy chart (120 epochs)

The next illustration represents the results of the colorization of a random batch from the validation set:

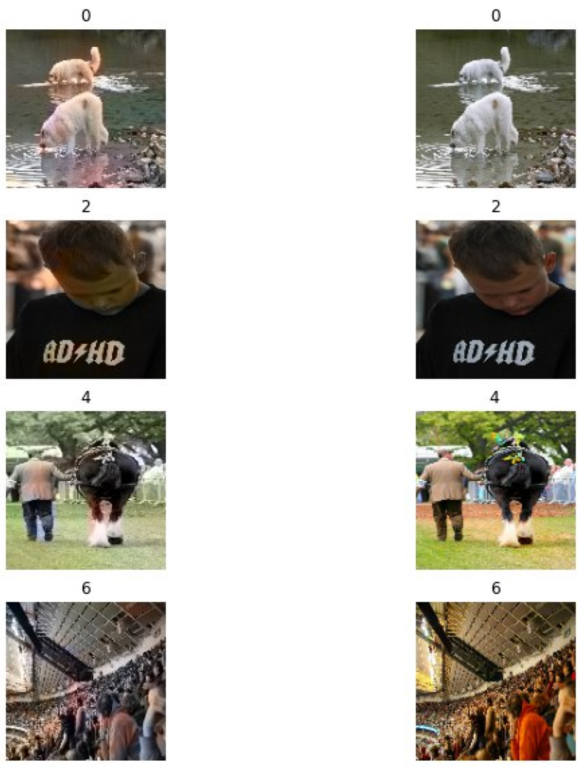


Figure 5: CNN Model - Colorization Results

7.2 Autoencoder Model

The results are:

- Train Loss: 92.0349 MSE
- Train Accuracy: 78.80%
- Validation Loss: 241.1622 MSE
- Validation Accuracy: 63.50%

The next illustration represents the results of the colorization of a random batch from the validation set:

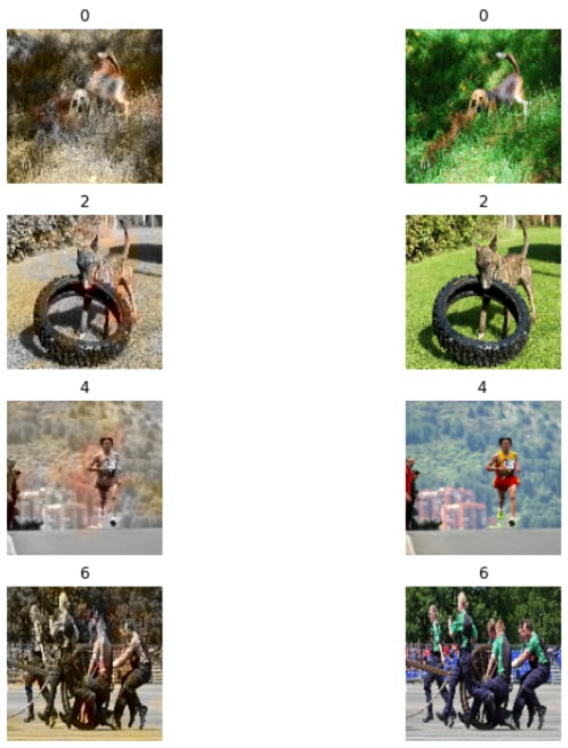
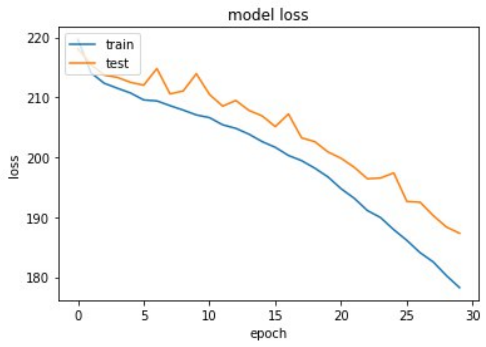


Figure 6: Autoencoder Model - Colorization Results

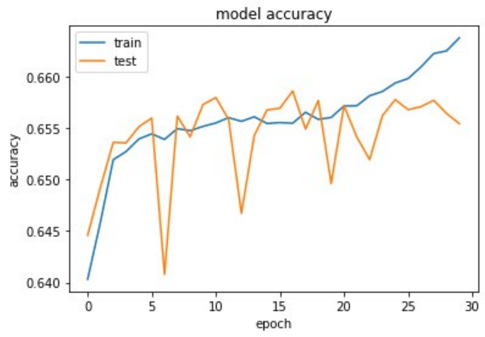
7.3 CNN using Pretrained Model

The results are:

- Train Loss: 130 MSE
- Train Accuracy: 70.0 %
- Validation Loss: 155 MSE
- Validation Accuracy: 68.7%



(a) CNN using Pretrained Model - Loss chart (30 epochs)



(b) CNN using Pretrained Model - Accuracy chart (30 epochs)

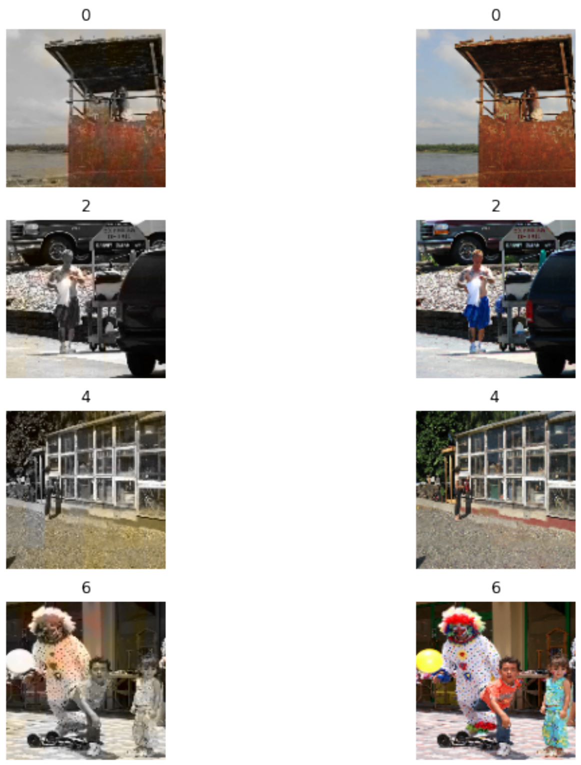


Figure 8: CNN using Pretrained Model - Colorization Results

References

[1] CIELAB color space. *Wikipedia Article: Advantages paragraph.*

[2] Flickr 30k dataset. *https://www.kaggle.com/adityajn105/flickr30k.*

[3] Flickr 8k dataset with captions. *https://www.kaggle.com/kunalgupta2616/flickr-8k-images-with-captions.*

[4] tf.data.dataset api. *https://www.tensorflow.org/api$_d$ocs/python/tf/data/Dataset.*

[5] Anwar, S., Tahir, M., Li, C., Mian, A., Khan, F. S., and Muzaffar, A. W. Image colorization: A survey and dataset, 2020.

[6] Chollet, F. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2017), pp. 1251–1258.

[7] Lewinson, E. Image colorization using convolutional autoencoders. *https://towardsdatascience.com/image-colorization-using-convolutional-autoencoders-fdabc1cb1dbe.*

[8] Zhang, R., Isola, P., and Efros, A. A. Colorful image colorization. In *European conference on computer vision* (2016), Springer, pp. 649–666.