Bilkent University
Faculty of Engineering
Computer Engineering Department

CS464 – Introduction to Machine Learning

Fall 2020

Final Report

Beril Bayram
Cem Daloğlu
Kenan Korhan Odabaş
Damla Kartal
Tuna Özcan

ID
numbers

I. Introduction

Image colorization has always been an important problem throughout history. It was traditionally reserved for artists with excellent abilities, while today, the black and white photographs can be colorized using Machine Learning algorithms with great accuracy [1]. For this project, grayscale images were colorized into their "lab" color channel versions. To do that, three different Convolutional Neural Network (CNN) algorithms were implemented, namely, Vanilla, ResNet, and U-Net. Vanilla CNN was considered as a baseline, and the other algorithms were compared according to this baseline. U-Net and ResNet models performed way better than the Vanilla CNN, which was chosen as a baseline. In the comparison between ResNet and U-Net, the ResNet model performed better than U-Net.

II. Problem Description

Gray scaled images are mostly known to be used before colorized images that we are familiar with today became popular. With the increasing technological advancements in imagery and photography, grayscale images are rarely used in cameras and smartphones and usually only available as filters. Yet, grayscale images have the advantage compared to the red, green, blue coded images when data size is a concern. RGB images need three times the storage for every image with the increased pixel density in current devices. RGB images can fill up the space forcing the user to delete their old images to take new ones. For this type of use case, it is possible to use image colorization algorithms where the storage can be saved for more images down the product's life cycle. Thus, the aim is to see whether grayscale images can be colored fully using Machine Learning. In addition to that, to see how different CNN algorithms perform on image colorization, three different CNN algorithms (Vanilla, ResNet, and U-Net) were implemented and compared.

III. Methods

A dataset was chosen to evaluate the problem, and this dataset includes grayscale and color versions of 25.000 224 x 224-pixel images [2]. The data collection does not include the colored images in the RGB color space; they reside in a color space of the LAB. There are three color channels in the LAB color space: The L channel encodes the strength of light from black (0) to white (100), the A channel encodes green (-) to red (+), and the B channel encodes blue (-) to yellow (+). Moreover, we know that the gray images consist of only one dimension, which displays shades of black or white, and do not include color detail instead of the color space of the LAB.

The selected dataset is categorized into three main categories: training set, validation set, and test set. Algorithms are trained in the training stage to transform each gray image pixel to the colored pixel of its LAB value. Briefly, each image's LAB color and gray values are mapped by the algorithm, and during the test process, an unseen gray image is used as an input while the model predicts the color image as output.

This project also aims to test numerous algorithms and to see which algorithm is the best for the colorization of the images. For this reason, three algorithms were implemented and compared. Convolutional Neural Network (CNN) Vanilla, ResNet, and U-Net were used as the primary algorithms. The baseline is selected for Vanilla CNN, and a comparison of the other algorithms will be made according to the performance efficiency of the Vanilla CNN algorithm. For the sake of performance, the transfer learning method was used in ResNet and U-Net algorithms.

The LAB data was divided into train, test, and validation data before the model implementation for each algorithm. Preprocessing is applied to the dataset to construct the desired model. The grayscale data was cast to type float32 and normalized by dividing the grayscale data by 255. Black and white images are eliminated on the colored dataset. Each model used the 'adam' optimizer.

Vanilla CNN Model Description

Various characteristics of images can be detected and learned by CNN algorithms that can have tens to hundreds of layers. Each layer learns to recognize multiple features of an image. At different resolutions, each image gets filtered, and the outputs are convolved with the input of the next layer. To be able to define the object uniquely, the filters can start with simple features like brightness or edges, and the complexity may increase throughout the process. CNN includes input, output, and hidden layers as the other implementations of neural networks.

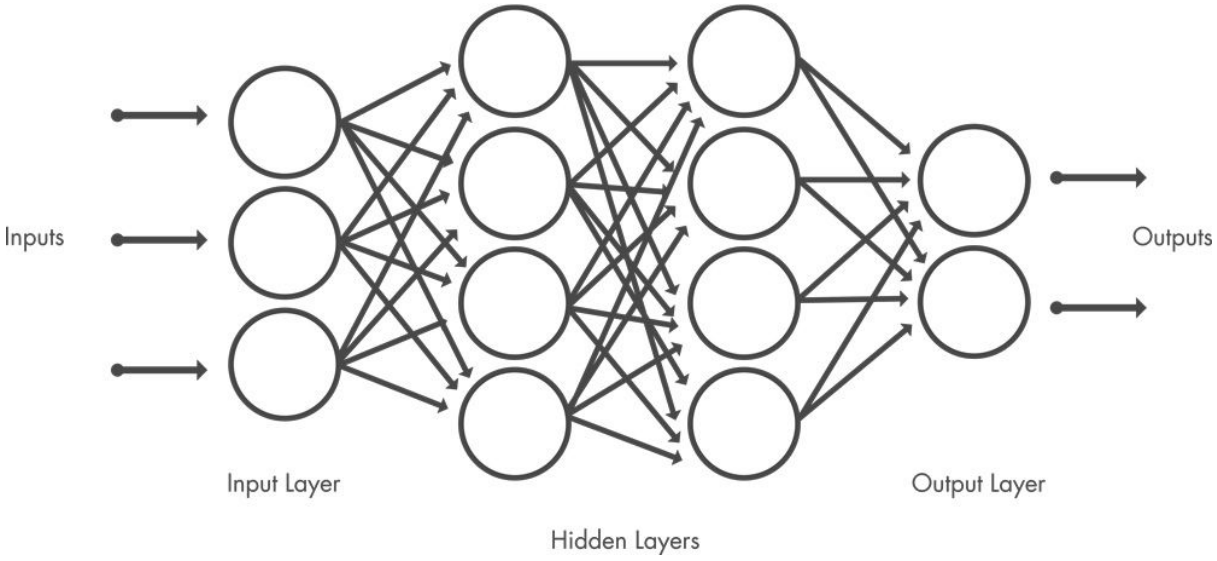


Figure 1: CNN input and output relationship [3].

The operations in these layers change the data in order to learn data-specific functions. The three commonly used layers are: convolution, activation or ReLU, and pooling.

Convolution: Images are going through convolutional filters, which activate certain features of the images.

Rectified linear unit (ReLU): It allows for more effective training by changing negative values to zero and keeping the positive values. ReLU carries only the activated features to the next layer.

Pooling: By nonlinear downsampling pooling reduces the parameters that the network needs and simplifies the output.

These layers are repeated many times to identify many features of the images [3].

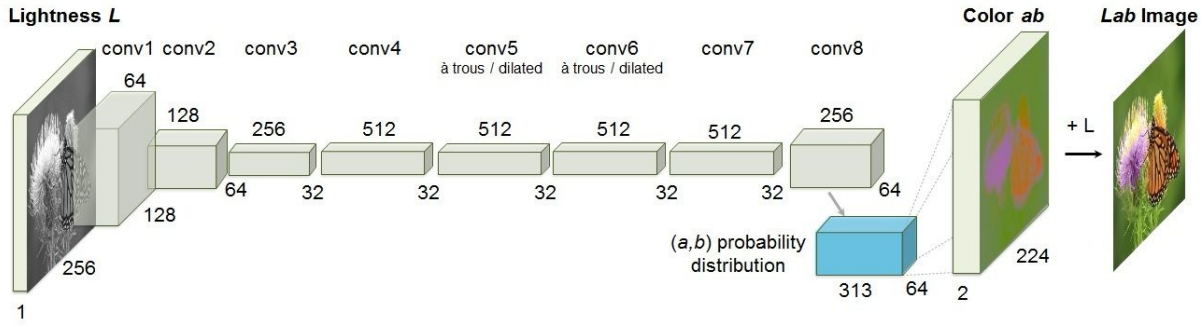


Figure 2: An image going through convolutional layers [1].

The CNN algorithm relies on convolutional layers to filter the specific features found on the gray scaled images. For this operation, the image is first examined in terms of lightness from 0 to 255, which corresponds to black and white. Conventionally colors are explained in terms of red, green, and blue color channels for image colorization tasks, we can also use green-red (a) and blue-yellow (b) channels. The human eye operates in a similar manner focussing on the lightness of the image before colorizing. We use the convolutional layers as a substitute to what human eye receptors do by the neural network; we try to recognize patterns and narrow them to two channels at the end, which ultimately leads to the 'a','b' color channels. The neural network in Figure 2 represents a 256x256 image going through convolutional layers with increasing filters and finally being represented as a "lab" image with the activation function.

For Vanilla CNN, the Keras API, which runs on TensorFlow, was used. Two different architectures were applied for Vanilla CNN, and the results of these architectures were compared in order to get better results. After the preprocessing procedure described above, the two models were constructed with different architecture.

For the first architecture, using the ReLU activation function, the 2D convolutional/transpose layers were activated, and these layers were applied with a different kernel and filter sizes. The model architecture can be seen in Table 1.

Model: "sequential"	
Layer (type)	**Output Shape**
conv2d (Conv2D)	(None, 222, 222, 32)
conv2d_1 (Conv2D)	(None, 222, 222, 16)
conv2d_transpose (Conv2DTran)	(None, 224, 224, 2)

Table 1: First Model Architecture

The architecture of the model can be seen in Table 2. For this model, the difference is that a 2D max-pooling layer was also applied in addition to 2D convolutional/transpose layers to increase the model's complexity.

Model: “sequential”	
Layer (type)	**Output Shape**
conv2d (Conv2D)	(None, 222, 222, 32)
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)
conv2d_1 (Conv2D)	(None, 109, 109, 16)
conv2d_transpose (Conv2DTran)	(None, 224, 224, 2)

Table 2: Second Model Architecture

For the second model, each pixel's color density was increased by a 2D max-pooling layer. For example, the second model, which has a 2D max-pooling layer, predicts the colors denser than the first model. For some cases, since using a 2D max-pooling layer can decrease the computational cost, it is beneficial.

ResNet Model Description

After the first implementation of CNN, each architecture started to use more layers to reduce the error. However, after some point the algorithm started to come up with a higher error rate which would not be the case if the algorithm’s problem was simply the overfitting. Researchers then realized that with each layer added, the deep learning algorithm came up with a problem called Vanishing/Exploding gradient. This problem happens when the number of layers yields gradients to be either zero or too big. Thus, the error rate increases with each layer. In the following Figure 3, effect of layer numbers on errors of testing and training sets can be observed:

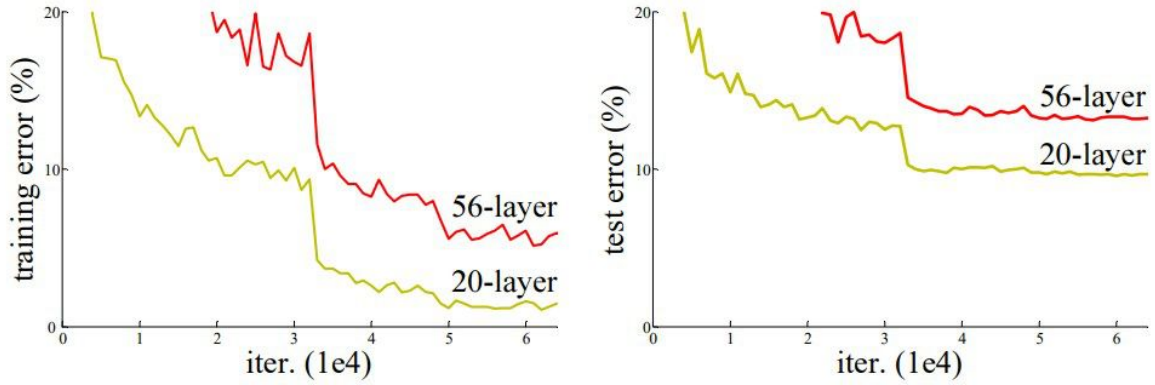


Figure 3: The relationship between test and train error with number of iterations[4].

Then in 2015 Microsoft Research proposed the new architecture called Residual Network (ResNet). This architecture was a proposed solution to the problem of vanishing/exploding gradient. For a residual network the approach taken into account is skipping the connections. Because if there exists any layer that affects the performance of the algorithm badly these layers will eventually be skipped by regularization and the performance

of the algorithm increases. Thus, when some of the training layers are skipped and connected to the output it came up with better results.

The implementation of the ResNet algorithm can be examined on Figure 4 below. First layer in the ResNet algorithm is the conventional convolution layer that is predetermined by the input images shape then the output of this convolutional layer is added on the output of the convolution layer that is constructed two layers after the initial layer, their activation functions are arranged according to this setup. For the 18-layer residual implementation the convolutional layer filters are doubled at every two layers leading to more parameters in terms of analyzing the features [4].

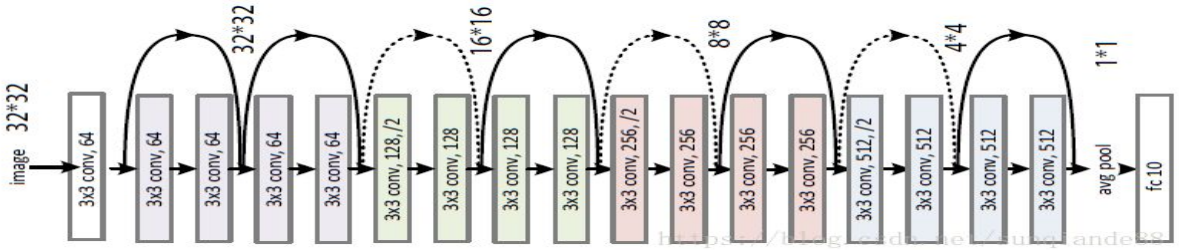


Figure 4: 18 Layer ResNet [4].

Note that we used Pytorch for the implementation of the ResNet model and also applied the same model by using Keras [5]. ResNet model architecture which is implemented by using Pytorch can be seen in Table 3. First half of the model includes the ResNet18 model which has 18 layers and the second half contains 14 layers for upsampling purposes. The model is trained with 4000 preprocessed training data and 800 of data is splitted as validation set. Training is applied with batch size equal to 32 and 250 epochs. 100 test data which are not in the training set is applied to the model and some successful results are shown in Figure 16, Figure 18 and in the Appendix section.

Layer (type)	Output Shape	Param
Conv2d-1	[32, 64, 112, 112]	3136
BatchNorm2d-2	[32, 64, 112, 112]	128
ReLU-3	[32, 64, 112, 112]	0
MaxPool2d-4	[32, 64, 56, 56]	0
Conv2d-5	[32, 64, 56, 56]	36864
BatchNorm2d-6	[32, 64, 56, 56]	128
ReLU-7	[32, 64, 56, 56]	0
Conv2d-8	[32, 64, 56, 56]	36864
BatchNorm2d-9	[32, 64, 56, 56]	128
ReLU-10	[32, 64, 56, 56]	0

BasicBlock-11	[32, 64, 56, 56]	0
Conv2d-12	[32, 64, 56, 56]	36864
BatchNorm2d-13	[32, 64, 56, 56]	128
ReLU-14	[32, 64, 56, 56]	0
Conv2d-15	[32, 64, 56, 56]	36864
BatchNorm2d-16	[32, 64, 56, 56]	128
ReLU-17	[32, 64, 56, 56]	0
BasicBlock-18	[32, 64, 56, 56]	0
Conv2d-19	[32, 128, 28, 28]	73728
BatchNorm2d-20	[32, 128, 28, 28]	256
ReLU-21	[32, 128, 28, 28]	0
Conv2d-22	[32, 128, 28, 28]	147456
BatchNorm2d-23	[32, 128, 28, 28]	256
Conv2d-24	[32, 128, 28, 28]	8192
BatchNorm2d-25	[32, 128, 28, 28]	256
ReLU-26	[32, 128, 28, 28]	0
BasicBlock-27	[32, 128, 28, 28]	0
Conv2d-28	[32, 128, 28, 28]	147456
BatchNorm2d-29	[32, 128, 28, 28]	256
ReLU-30	[32, 128, 28, 28]	0
Conv2d-31	[32, 128, 28, 28]	147456
BatchNorm2d-32	[32, 128, 28, 28]	256
ReLU-33	[32, 128, 28, 28]	0
BasicBlock-34	[32, 128, 28, 28]	0
Conv2d-35	[32, 128, 28, 28]	147584
BatchNorm2d-36	[32, 128, 28, 28]	256
ReLU-37	[32, 128, 28, 28]	0
Upsample-38	[32, 128, 56, 56]	0
Conv2d-39	[32, 64, 56, 56]	73792
BatchNorm2d-40	[32, 64, 56, 56]	128

ReLU-41	[32, 64, 56, 56]	0
Conv2d-42	[32, 64, 56, 56]	36928
BatchNorm2d-43	[32, 64, 56, 56]	128
ReLU-44	[32, 64, 56, 56]	0
Upsample-45	[32, 64, 112, 112]	0
Conv2d-46	[32, 32, 112, 112]	18464
BatchNorm2d-47	[32, 32, 112, 112]	64
ReLU-48	[32, 32, 112, 112]	0
Conv2d-49	[32, 2, 112, 112]	578
Upsample-50	[32, 2, 224, 224]	0

Table 3: ResNet architecture.

U-Net Model Description

The U-Net algorithm was developed along with the increasing popularity of the neural networks in Biomedical Image Segmentation by Olaf Rennobergfer. The designed algorithm is constructed by two paths named as the contraction path which is also known as the encoder path and the symmetric expanding path which can be called the decoder of the algorithm. The encoding algorithm consists of convolutional and max pooling layers that are added to the base sequential model. The decoding layer allows the localization of the data using transposed convolutional layers. With the two parts an end-to-end fully convolutional network (FCN) is constructed. Since the model contains only convolutional layers, input shape can be formed for any given image shape.

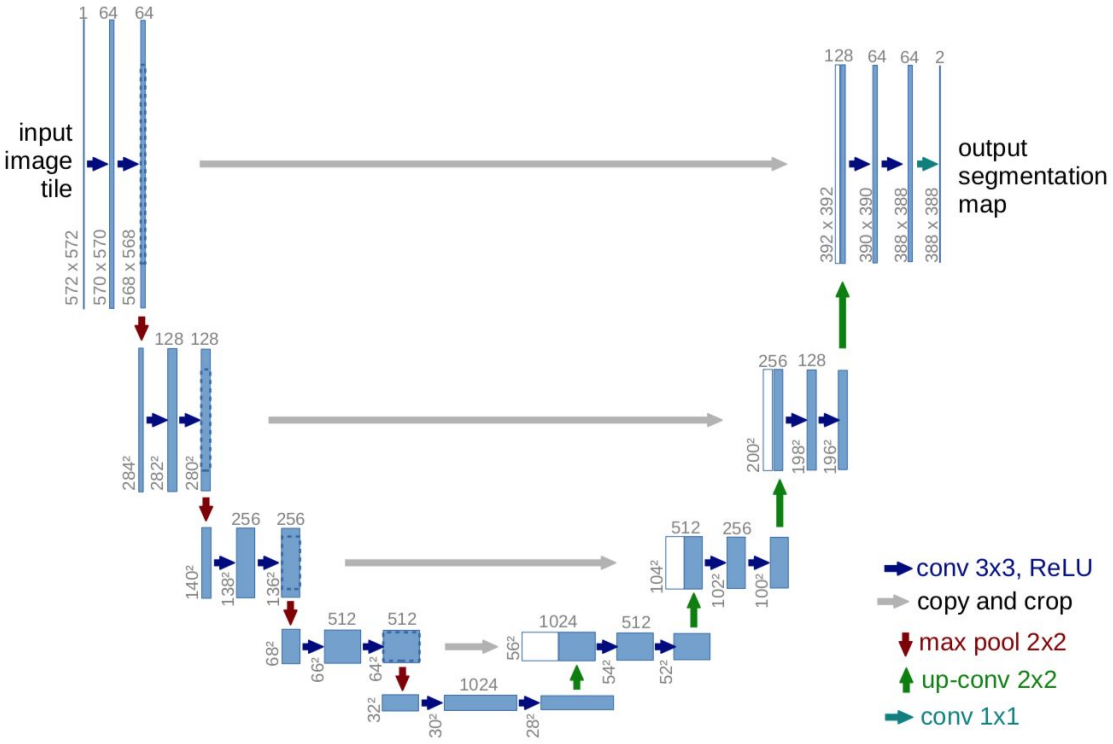


Figure 5: Preliminary model of U-Net [5].

Given the preliminary model in Figure 5 The blue indicators are the convolutional layer outputs that are first used by the encoder paths that use the traditional convolve-max pool strategy that we have discussed on Vanilla CNN. The encoder layer on the example ends up with 32x32 images at the lowest resolution for determining effective features. The lowest resolution images are then up-sampled with the recursive approach in which the previously encoded outputs are copied on the decoding stage represented as white boxes. The up sampling process continues with the neural network finding the output with the same shape as the input ending to end the output segmentation layers by using convolutional layers with decreasing filter size [6]. Different from the U-Net model above, our model has input shape of (224,224,1) and output shape of (224,224,2).

Our U-Net architecture is given below in Table 4. We used Keras for the implementation of the U-net model [7]. The model is trained with 4000 preprocessed training data and 800 of data is splitted as validation set. Training is applied with batch size equal to 16 and 250 epochs. 100 test data which are not in the training set is applied to the model and some successful results are shown in Figure 11, Figure 13 and in the Appendix section.

Layer (type)	Output Shape	Param	Connected to
input_1 (InputLayer)	(None, 224, 224, 1)	0	None
conv2d (Conv2D)	(None, 224, 224, 32)	320	input_1[0][0]
conv2d_1 (Conv2D)	(None, 224, 224, 32)	9248	conv2d[0][0]

max_pooling2d (MaxPooling2D)	(None, 112, 112, 32)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 112, 112, 64)	18496	max_pooling2d[0][0]
conv2d_3 (Conv2D)	(None, 112, 112, 64)	36928	conv2d_2[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 56, 56, 64)	0	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 56, 56, 128)	73856	max_pooling2d_1[0][0]
conv2d_5 (Conv2D)	(None, 56, 56, 128)	147584	conv2d_4[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 28, 28, 128)	0	conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 28, 28, 256)	295168	max_pooling2d_2[0][0]
conv2d_7 (Conv2D)	(None, 28, 28, 256)	590080	conv2d_6[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 14, 14, 256)	0	conv2d_7[0][0]
conv2d_8 (Conv2D)	(None, 14, 14, 512)	1180160	max_pooling2d_3[0][0]
conv2d_9 (Conv2D)	(None, 14, 14, 512)	2359808	conv2d_8[0][0]
conv2d_transpose (Conv2D_Transpose)	(None, 28, 28, 256)	524544	conv2d_9[0][0]
concatenate (Concatenate)	(None, 28, 28, 512)	0	conv2d_transpose[0][0], conv2d_7[0][0]
conv2d_10 (Conv2D)	(None, 28, 28, 256)	1179904	concatenate[0][0]
conv2d_11 (Conv2D)	(None, 28, 28, 256)	590080	conv2d_10[0][0]
conv2d_transpose_1 (Conv2DTranspose)	(None, 56, 56, 128)	131200	conv2d_11[0][0]
concatenate_1 (Concatenate)	(None, 56, 56, 256)	0	conv2d_transpose_1[0][0], conv2d_5[0][0]
conv2d_12 (Conv2D)	(None, 56, 56, 128)	295040	concatenate_1[0][0]
conv2d_13 (Conv2D)	(None, 56, 56, 128)	147584	conv2d_12[0][0]
conv2d_transpose_2 (Conv2DTranspose)	(None, 112, 112, 64)	32832	conv2d_13[0][0]
concatenate_2 (Concatenate)	(None, 112, 112, 128)	0	conv2d_transpose_2[0][0], conv2d_3[0][0]
conv2d_14 (Conv2D)	(None, 112, 112, 64)	73792	concatenate_2[0][0]
conv2d_15 (Conv2D)	(None, 112, 112, 64)	36928	conv2d_14[0][0]
conv2d_transpose_3 (Conv2DTranspose)	(None, 224, 224, 32)	8224	conv2d_15[0][0]
concatenate_3 (Concatenate)	(None, 224, 224, 64)	0	conv2d_transpose_3[0][0], conv2d_1[0][0]

conv2d_16 (Conv2D)	(None, 224, 224, 32)	18464	concatenate_3[0][0]
conv2d_17 (Conv2D)	(None, 224, 224, 32)	9248	conv2d_16[0][0]
conv2d_18 (Conv2D)	(None, 224, 224, 2)	578	conv2d_17[0][0]

Table 4: U-NET Architecture.

For comparing the predicted image and the ground truth, addition to subjective comparison, two more image quality metrics were used. The first one is Peak Signal to Noise Ratio (pSNR) which can be calculated using the Equation 1:

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right)$$

Equation 1: Equation for pSNR [8].

The second one is Structural Similarity (SSIM) Index which can be calculated using the Equation 2:

$$\mathrm{SSIM}(x, y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

Equation 2: Equation for SSIM [8].

SSIM will return a number between 0 and 1. If the number is closer to 1, the predicted image will be more similar to ground truth. For calculating the SSIM, skimage.measure method was used and for calculating pSNr, cv2.PSNR method was used.

IV. Results

Prediction Results of Vanilla CNN Model

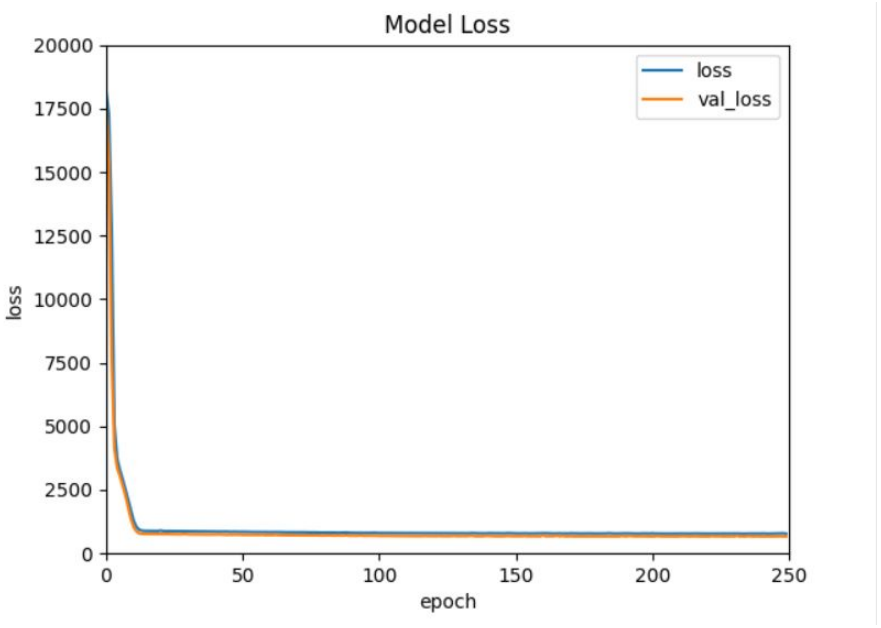


Figure 6: Vanilla Train and Validation MSE Loss

As it can be seen from the figure 6, MSE loss of Vanilla CNN model decreases as we apply more epochs, and test MSE loss converges to 700 approximately. This convergence shows that our Vanilla CNN model finishes its learning process after 50 epochs.

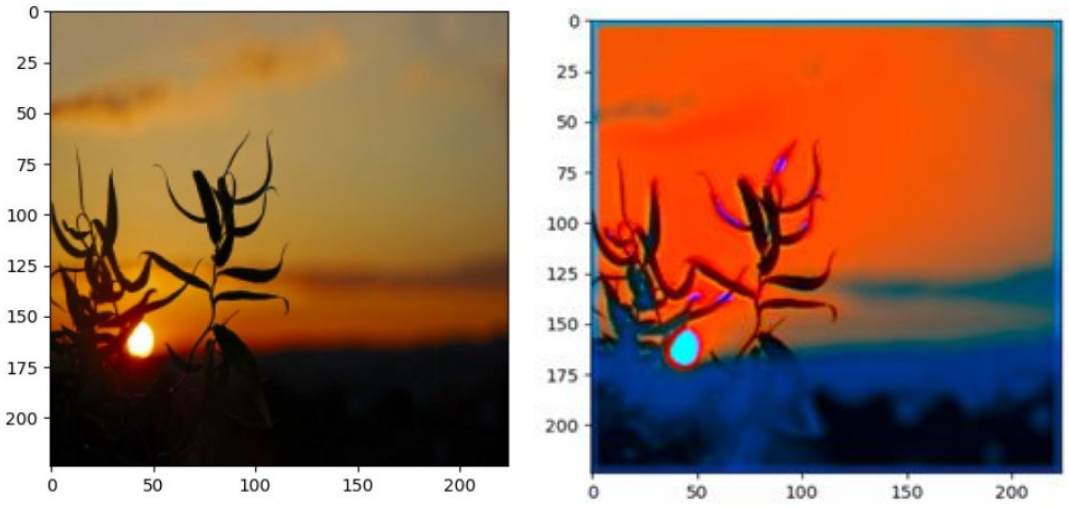


Figure 7 and 8: Actual Image vs Predicted Colorized Image

SSIM: 0.573

PSNR: 9.548

Results of U-Net Model

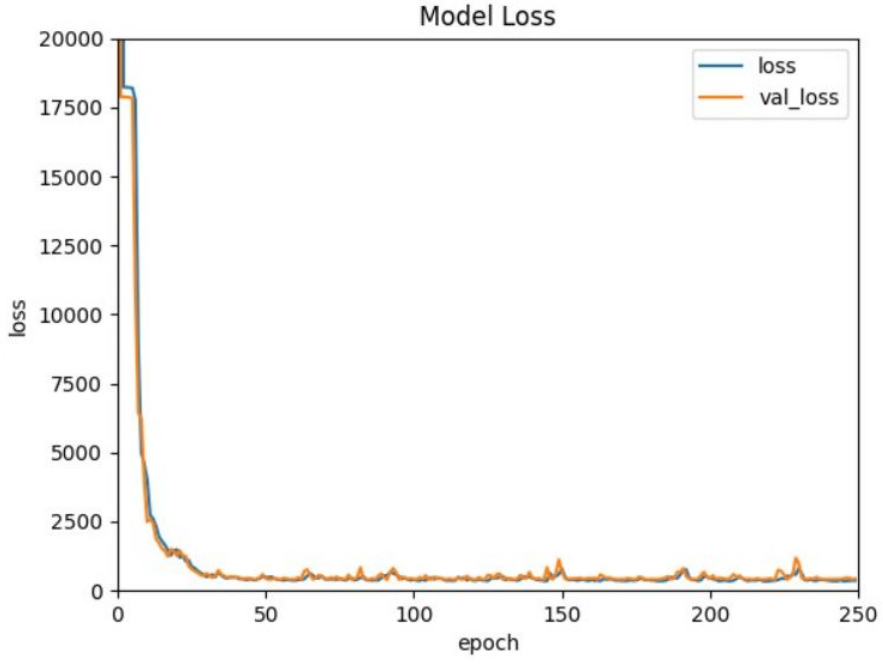


Figure 9: U-Net Train and Validation MSE Loss

As it can be seen from the figure 9, MSE loss of U-Net model decreases as we apply more epochs, and test MSE loss converges to 220 approximately. This convergence shows that our U-Net model finishes its learning process after 250 epochs.

Successful Prediction Results of U-Net Model

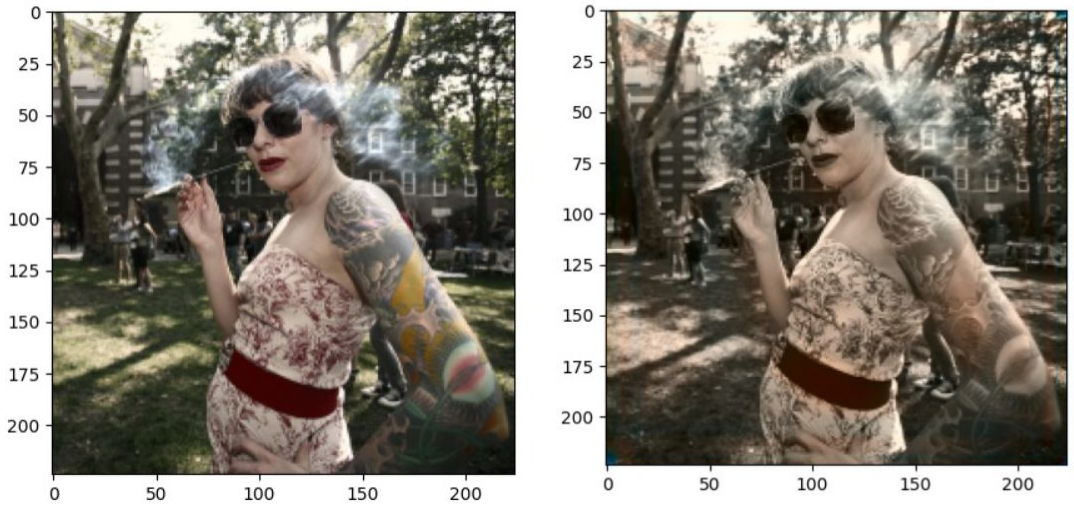


Figure 10 and 11: Actual Image vs Predicted Colorized Image

SSIM: 0.4757
PSNR: 11.659

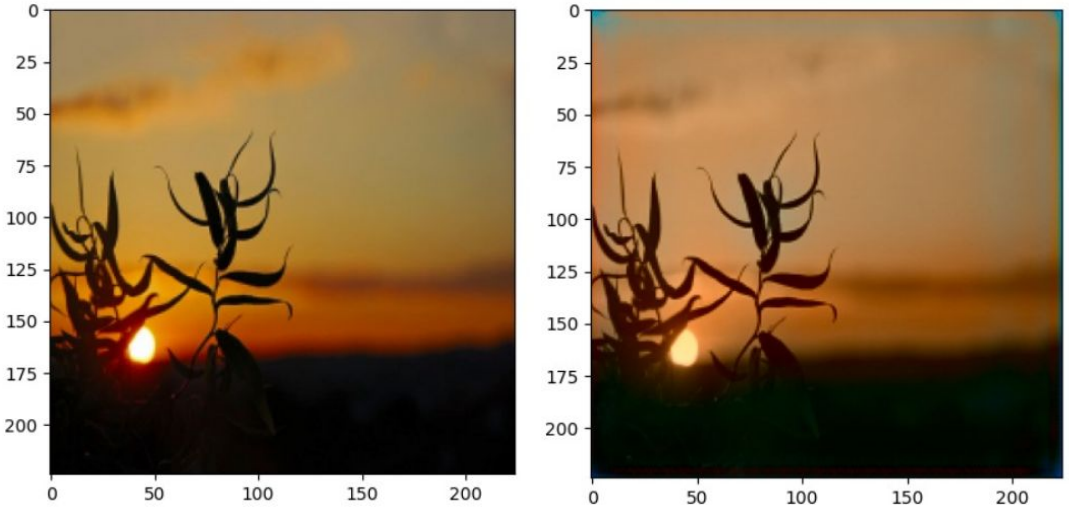


Figure 12 and 13: Actual Image vs Predicted Colorized Image

SSIM: 0.573
PSNR: 9.548

Results of ResNet Model

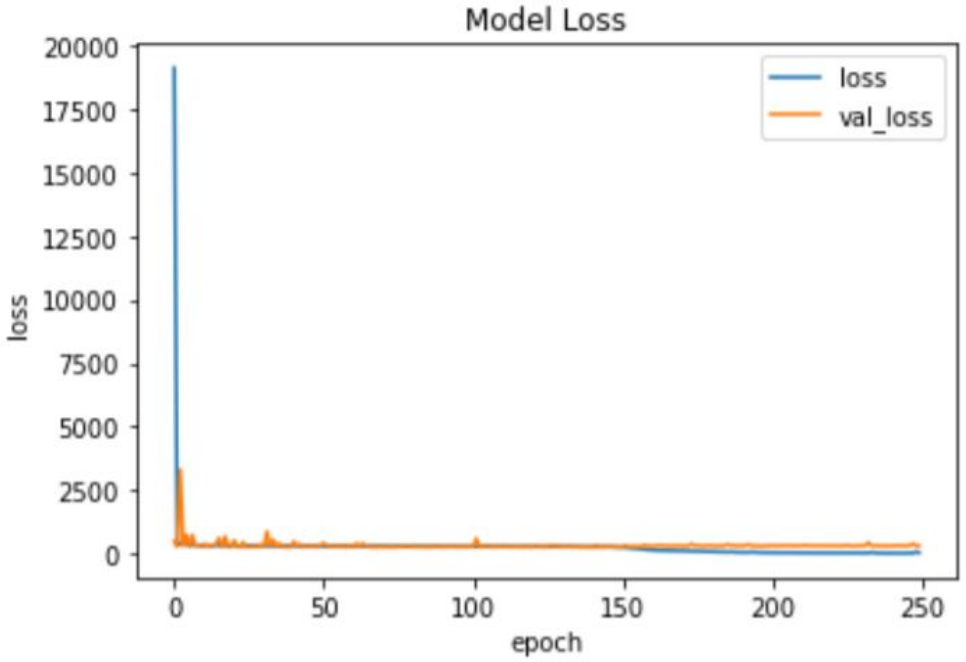


Figure 14: ResNet Train and Test MSE Loss

As it is seen from the figure 14, MSE loss of the ResNet model decreases as we apply more epochs, and test MSE loss converges to 50 approximately. This convergence shows that our ResNet model finishes its learning process after 250 epochs.

Successful Prediction Results of ResNet Model

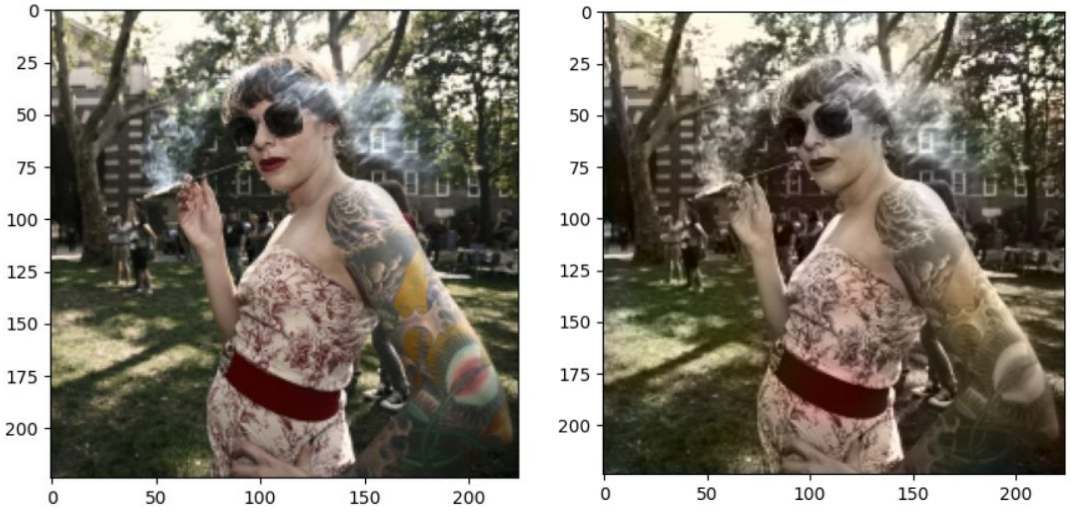


Figure 15 and 16: Actual Image vs Predicted Colorized Image

SSIM: 0.4757
PSNR: 11.659

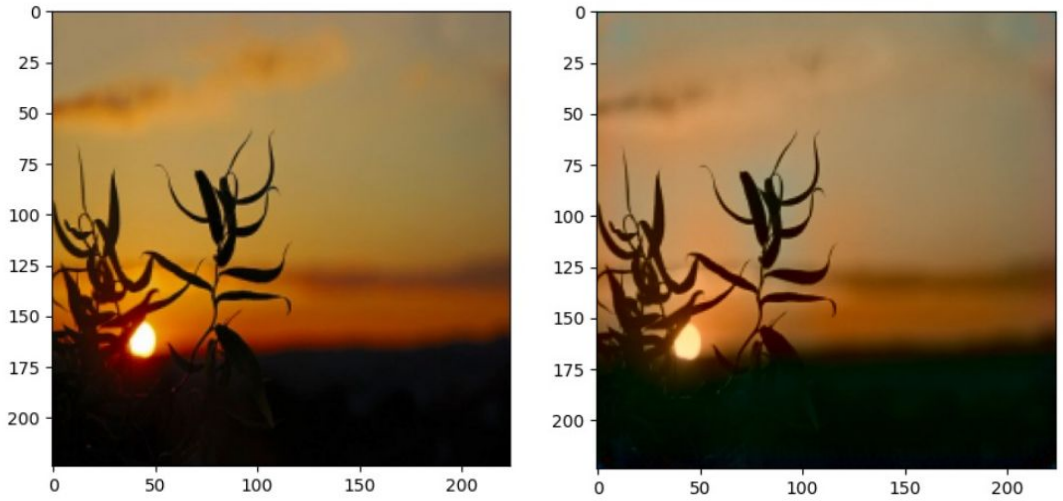


Figure 17 and 18: Actual Image vs Predicted Colorized Image

SSIM: 0.573
PSNR: 9.548

Unsuccessful Prediction Results of ResNet Model

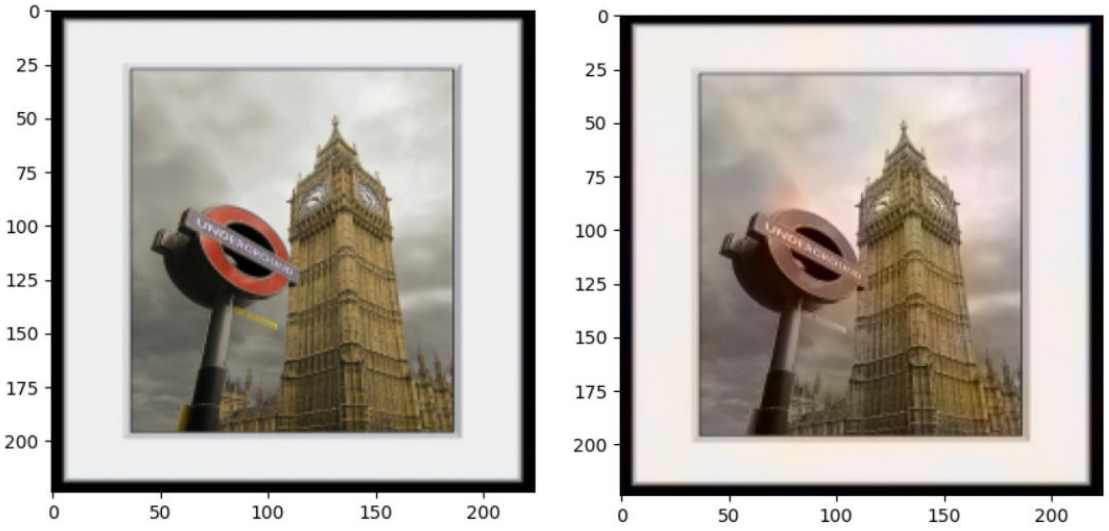


Figure 19 and 20: Actual Image vs Predicted Colorized Image

Unsuccessful Prediction Results of U-Net Model

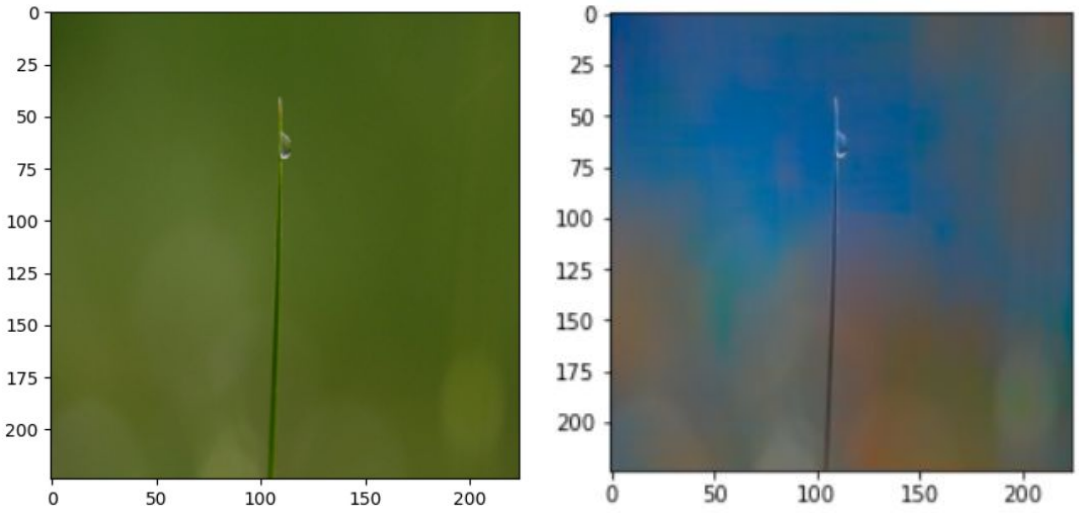


Figure 21 and 22: Actual Image vs Predicted Colorized Image

More prediction results (both successful and unsuccessful) of U-Net, ResNet and Vanilla models can be seen in the Appendix section.

V. Discussion

By looking at Figure 8, Figure 13 and Figure 18, it could be said that the ResNet model is better than the U-Net and U-Net is better than Vanilla CNN model. ResNet and U-Net models can be compared with Figure 11 and Figure 16 as well. In both images, ResNet prediction is more similar to the original image than U-Net model predictions. The subjective results are also supported by mathematical MSE loss metric. By looking at Figure 6, Figure 9 and Figure 14, the ResNet model converged to the lowest MSE loss value compared to the other models and the U-Net model converged to the second lowest MSE loss. Since all models' SSIM and PSNR values are the same, it could be said that none of the models add any noise or blurriness to the predictions.

While observing predictions, we saw that some of the images at the colored data are grayscale. These grayscale images increase the weights of black and white colors because the model gets high accuracy results on validation part and so all predictions result with a gray tone. Thus, we decided to implement a preprocessor and eliminate the black&white images on the dataset's colored part. After preprocessing, gray tone is eliminated in the predictions but this time a different color (mostly yellow or brown) tone is observed. This is caused by images which have the same a and b value in every pixel. These images also gave high accuracy results so weighs are modified with respect to these colors. Moreover, normalization is applied to the input data. Since the input L can take values between 0 and 255, the model cannot pick up the contribution of the smaller magnitude features. Theoretically, L(lightness) value of a LAB image should be in the range between 0 and 100. However, we realized that it exists in the range between 0 and 255 in our dataset. In the implementation, this theoretical mismatch did not cause any problem, that's why we ignored that.

Some of the above images are our good predictions but not all of the predictions are that good. The dataset that we trained our models contain mostly images with sunlight. Therefore, each model's weights are dominated by yellow or brown color weights which cause predictions to have brown tone overall. These weights gave better predictions for images with sun and sunlight but for images without yellow or brown color it causes mispredictions. For this dataset, the U-Net model less accurately learned the green color than the ResNet model. Other unsuccessful U-Net results can also be seen in the Appendix section. For the ResNet model, there is no specific color that the model could not learn. However, some test images are unsuccessfully predicted by the ResNet model, they can be seen in the Appendix section. To generalize, for the same pictures ResNet model gave better results than U-Net and ResNet model has lower number of unsuccessful predictions than U-Net model.

VI. Conclusions

Our aim was to colorize grayscale images, as it is stated in the problem description section, and after implementing the described methods we achieved our goal. Grayscale images can be colored mostly but not fully.

We learned that the ResNet model has better accuracy overall images than U-Net and Vanilla models. For our dataset, the ResNet model can predict more colors than other models so it is applicable to colorize grayscale images. U-Net model is also applicable for colorization for some grayscale images especially the images with sunlight. However, the U-net model gave unsuccessful predictions for green color for the applied dataset. The same U-net model may give better predictions for green with different dataset. Lastly, the Vanilla model is not applicable since it could not colorize a grayscale image detailly.

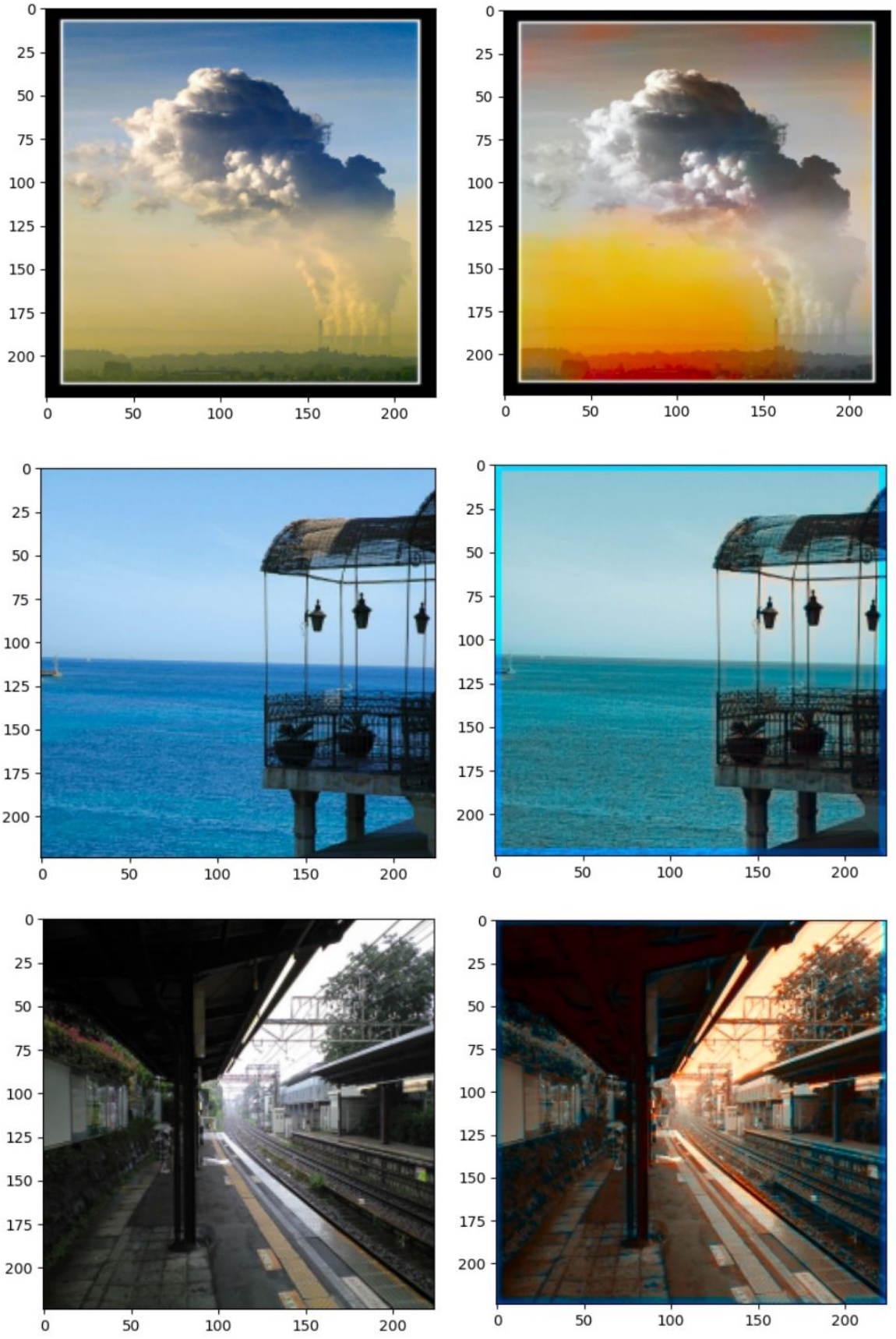
To sum up, the ResNet model is the most preferable model among the other models for such an Image Colorization task and with the dataset we used.

VII. Appendix

Kenan Korhan Odabaş – Design and Implementation of ResNet and Vanilla algorithms, report writing, building NVIDIA GPU Computing Tool CUDA, preparing the presentation.
Cem Daloğlu – Design and Implementation design of U-Net and Vanilla algorithm, report writing, building NVIDIA GPU Computing Tool CUDA, preparing the presentation.
Tuna Özcan – Building NVIDIA GPU Computing Tool CUDA, report writing, preparing the presentation, researching, training and improvement of the algorithms.
Beril Bayram – Report writing, preparing the presentation, researching, training and improvement of the algorithms.
Damla Kartal – Building NVIDIA GPU Computing Tool CUDA, report writing, preparing the presentation, researching, training and improvement of the algorithms.

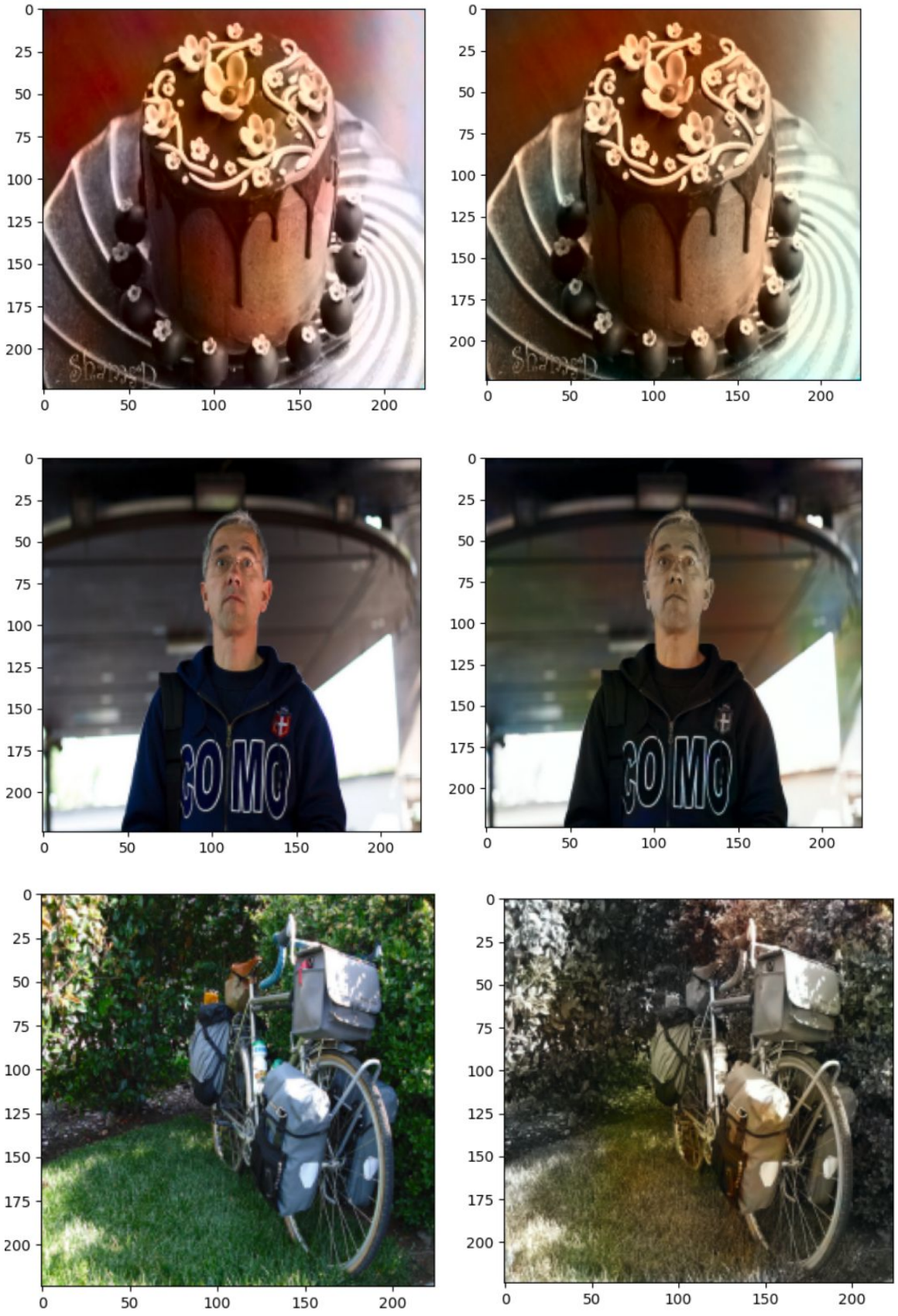
Prediction Results of Vanilla Models

Actual Image vs Predicted Image



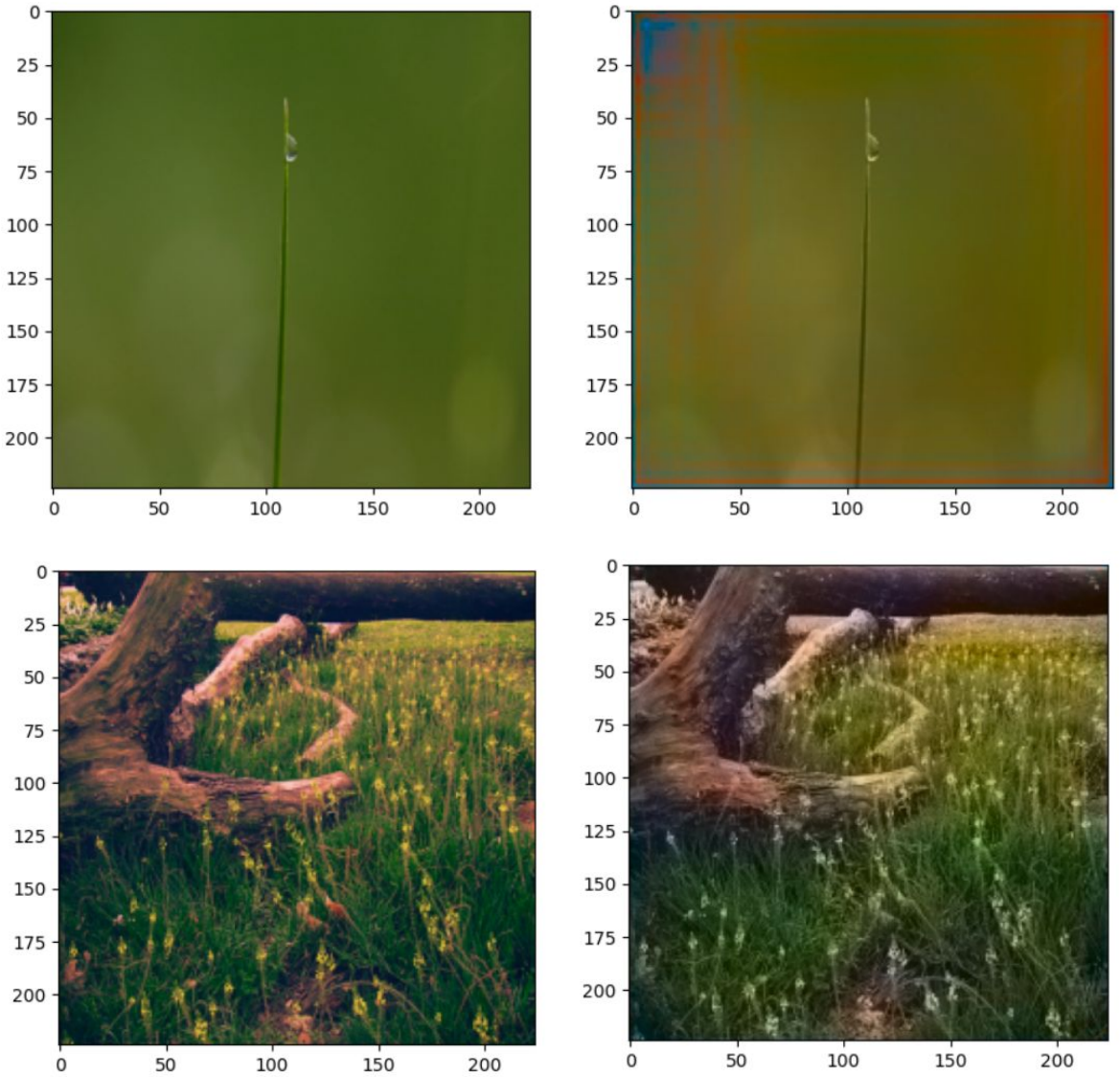
Prediction Results of U-Net Model

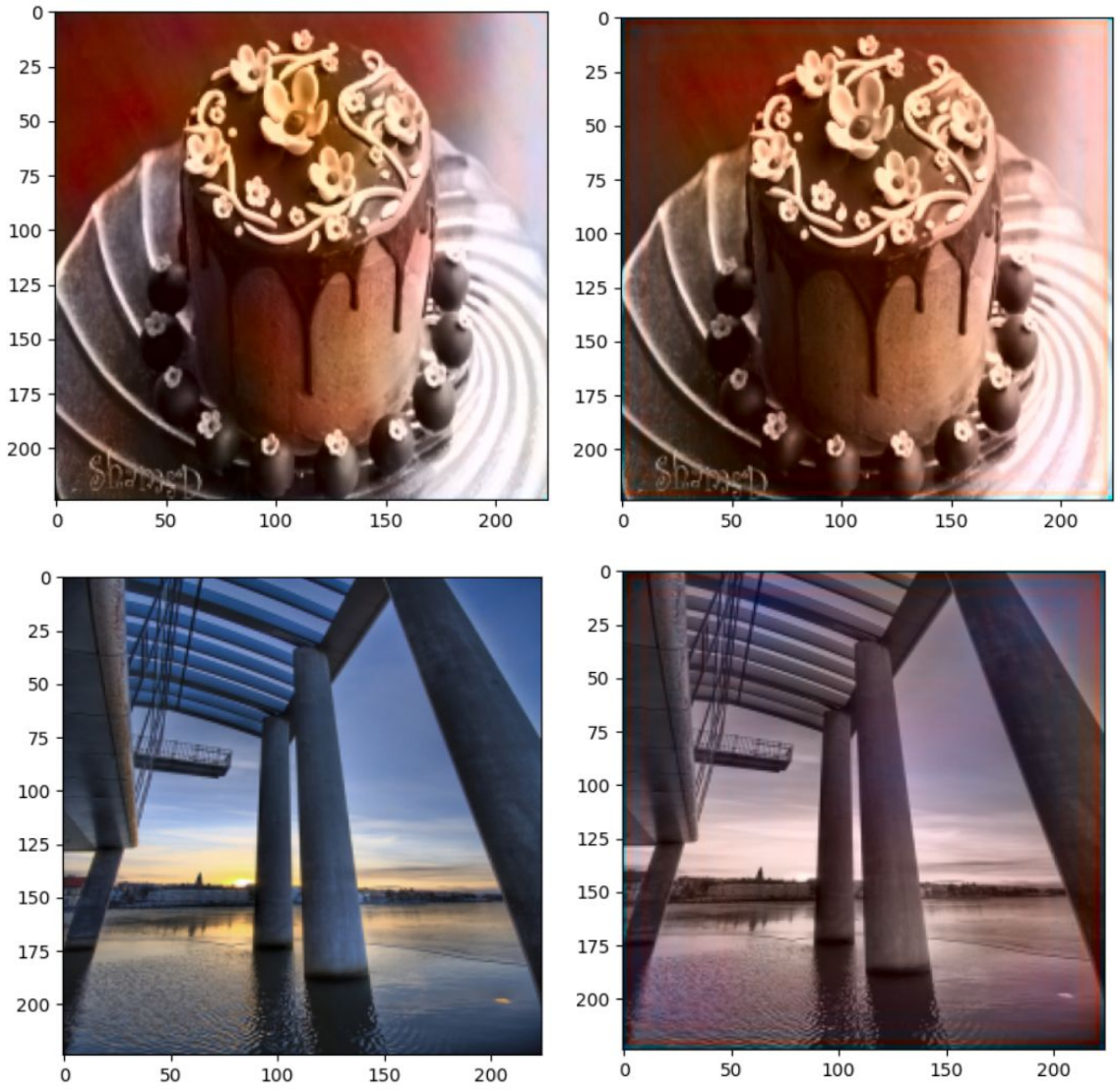
Actual Image vs Predicted Image



Prediction Results of ResNet Model

Actual Image vs Predicted Image





Vanilla CNN Python Code

```
# Libraries
import csv
import matplotlib.pyplot as plt
import cv2
import tensorflow as tf
import keras
from skimage.measure import compare_ssim
from tensorflow.keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten,
BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, InputLayer,
UpSampling2D, Conv2DTranspose, LeakyReLU, AveragePooling2D, \
```

```
   GlobalAveragePooling2D, GlobalMaxPooling2D, Input,
RepeatVector, Reshape, concatenate
import matplotlib.pyplot as plt
import numpy as np


images_gray = np.load("gray_scale.npy")
images_lab = np.load("ab1.npy")


# gray_scale
size_train = 4000
size_test = 100
size_test_begin = 0
change_pic = 0
X_train_l = images_gray[0+change_pic:change_pic+size_train]
X_train_l = X_train_l.reshape(size_train, 224, 224, 1)
X_test_l = images_gray[size_test_begin +
size_train+change_pic:change_pic+size_train + size_test_begin +
size_test]
X_test_l = X_test_l.reshape(size_test, 224, 224, 1)

# colored
X_train_lab = images_lab[0+change_pic:change_pic+size_train]
X_train_lab = X_train_lab.reshape(size_train, 224, 224, 2)
X_test_lab = images_lab[size_test_begin +
size_train+change_pic:change_pic+size_train + size_test_begin +
size_test]
X_test_lab = X_test_lab.reshape(size_test, 224, 224, 2)

X_gray_chick = np.zeros((1,224,224,2), dtype=float)
X_gray_chick = X_gray_chick.__add__(128)
check_test = 0
X_test_lab_temp = X_test_lab
X_test_l_temp = X_test_l
while check_test < size_test:
   if sum(sum(sum(sum(X_gray_chick ==
X_test_lab_temp[check_test])))/100352 > 0.4:
       X_test_l = np.zeros((size_test-1, 224, 224, 1),
dtype=float)
       X_test_l[0:check_test] = X_test_l_temp[0:check_test]
       X_test_l[check_test:] = X_test_l_temp[(check_test+1):]
```

```
        X_test_l_temp = X_test_l

        X_test_lab = np.zeros((size_test-1, 224, 224, 2),
dtype=float)
        X_test_lab[0:check_test] = X_test_lab_temp[0:check_test]
        X_test_lab[check_test:] = X_test_lab_temp[check_test + 1:]
        X_test_lab_temp = X_test_lab
        size_test = size_test-1
        check_test = check_test - 1
    check_test = check_test + 1

check_train = 0
X_train_lab_temp = X_train_lab
X_train_l_temp = X_train_l
while check_train < size_train:
    if sum(sum(sum(sum(X_gray_chick ==
X_train_lab_temp[check_train])))/100352 > 0.4:
        X_train_l = np.zeros((size_train-1, 224, 224, 1),
dtype=float)
        X_train_l[0:check_train] = X_train_l_temp[0:check_train]
        X_train_l[check_train:] = X_train_l_temp[(check_train+1):]
        X_train_l_temp = X_train_l

        X_train_lab = np.zeros((size_train-1, 224, 224, 2),
dtype=float)
        X_train_lab[0:check_train] =
X_train_lab_temp[0:check_train]
        X_train_lab[check_train:] = X_train_lab_temp[check_train +
1:]
        X_train_lab_temp = X_train_lab
        size_train = size_train-1
        check_train = check_train - 1
    check_train = check_train + 1

X_test_l = X_test_l/255
X_train_l = X_train_l/255

X_train_l = tf.cast(X_train_l, tf.float32)
X_test_l = tf.cast(X_test_l, tf.float32)

# #Vanilla CNN Model 1
# model = Sequential()
```

```
#
model.add(Conv2D(strides=1,filters=32,kernel_size=3,activation='re
lu',use_bias=True,padding='valid'))
#
model.add(Conv2D(strides=1,filters=16,kernel_size=3,activation='re
lu',use_bias=True,padding='valid'))
#
model.add(Conv2DTranspose(strides=1,filters=2,kernel_size=5,activa
tion='relu',use_bias=True,padding='valid'))

#Vanilla CNN Model 2
model = Sequential()
model.add(Conv2D(strides=1,filters=32,kernel_size=3,activation='re
lu',use_bias=True,padding='valid'))
model.add(MaxPooling2D(pool_size=(2,2),strides=2))
model.add(Conv2D(strides=1,filters=16,kernel_size=3,activation='re
lu',use_bias=True,padding='valid'))
model.add(Conv2DTranspose(strides=2,filters=2,kernel_size=8,activa
tion='relu',use_bias=True,padding='valid'))

model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
history = model.fit(x=X_train_l, y=X_train_lab,
validation_split=0.2, epochs=250, batch_size=16)
model.summary()
xx = model.predict(X_test_l)
xx = xx.reshape(size_test, 224, 224, 2)

h = history
plt.plot(h.history['loss'])
plt.plot(h.history['val_loss'])
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('Model Loss')
plt.legend(['loss', 'val_loss'])
plt.axis([0, 250, 0, 20000])
plt.show()
plt.plot(h.history['accuracy'])
plt.plot(h.history['val_accuracy'])
plt.legend(['accuracy', 'val_accuracy'])
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Model Accuracy')
```

```
plt.show()
for k in range(size_test):
   # predicted image
   img = np.zeros((224, 224, 3))
   kor = X_test_l[k]*255

   kor = kor[:, :, 0]
   img[:, :, 1:] = xx[k]
   img[:, :, 0] = kor

   img = img.astype('uint8')
   img_ = cv2.cvtColor(img, cv2.COLOR_LAB2RGB)
   plt.imshow(img_)
   plt.show()

   # actual image
   img_truth = np.zeros((224, 224, 3))
   kor_truth = X_test_l[k]*255
   kor_truth = kor_truth[:, :, 0]
   img_truth[:, :, 0] = kor_truth
   img_truth[:, :, 1:] = X_test_lab[k]

   img_truth = img_truth.astype('uint8')
   img_truth_ = cv2.cvtColor(img_truth, cv2.COLOR_LAB2RGB)
   plt.imshow(img_truth_)
   plt.show()
   (score, diff) = compare_ssim(img_truth, img_truth_, full=True,
multichannel=True)
   diff = (diff * 255).astype("uint8")
   print("SSIM: {}".format(score))
   print("PSNR: {}".format(cv2.PSNR(img_truth, img_truth_)))
   m = tf.keras.metrics.Accuracy()
   m.update_state(img_truth, img_truth_)
   print("Accuracy: ", m.result().numpy())
```

Code Block 1: Python code for Vanilla CNN.

U-Net Python Code

```
#Reference: The U-Net model is taken from the following link
https://www.kaggle.com/rahuldshetty/image-colorization-with-unet-a
uto-encoders/notebook
# Libraries
import csv
import matplotlib.pyplot as plt
import cv2
import tensorflow as tf
import keras
from skimage.measure import compare_ssim
from tensorflow import keras
from tensorflow.keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten,
BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, InputLayer,
UpSampling2D, Conv2DTranspose, LeakyReLU, AveragePooling2D, \
   GlobalAveragePooling2D, GlobalMaxPooling2D, Input,
RepeatVector, Reshape, concatenate
import matplotlib.pyplot as plt
import numpy as np
from keras.models import Model


images_gray = np.load("gray_scale.npy")
images_lab = np.load("ab1.npy")


# gray_scale
size_train = 4000
size_test = 100
size_test_begin = 0
change_pic = 0
X_train_l = images_gray[0+change_pic:change_pic+size_train]
X_train_l = X_train_l.reshape(size_train, 224, 224, 1)
X_test_l = images_gray[size_test_begin +
size_train+change_pic:change_pic+size_train + size_test_begin +
size_test]
X_test_l = X_test_l.reshape(size_test, 224, 224, 1)

# colored
X_train_lab = images_lab[0+change_pic:change_pic+size_train]
```

```
X_train_lab = X_train_lab.reshape(size_train, 224, 224, 2)
X_test_lab = images_lab[size_test_begin +
size_train+change_pic:change_pic+size_train + size_test_begin +
size_test]
X_test_lab = X_test_lab.reshape(size_test, 224, 224, 2)

X_gray_chick = np.zeros((1,224,224,2), dtype=float)
X_gray_chick = X_gray_chick.__add__(128)
check_test = 0
X_test_lab_temp = X_test_lab
X_test_l_temp = X_test_l
while check_test < size_test:
   if sum(sum(sum(sum(X_gray_chick ==
X_test_lab_temp[check_test])))/100352 > 0.4:
       X_test_l = np.zeros((size_test-1, 224, 224, 1),
dtype=float)
       X_test_l[0:check_test] = X_test_l_temp[0:check_test]
       X_test_l[check_test:] = X_test_l_temp[(check_test+1):]
       X_test_l_temp = X_test_l

       X_test_lab = np.zeros((size_test-1, 224, 224, 2),
dtype=float)
       X_test_lab[0:check_test] = X_test_lab_temp[0:check_test]
       X_test_lab[check_test:] = X_test_lab_temp[check_test + 1:]
       X_test_lab_temp = X_test_lab
       size_test = size_test-1
       check_test = check_test - 1
   check_test = check_test + 1

check_train = 0
X_train_lab_temp = X_train_lab
X_train_l_temp = X_train_l
while check_train < size_train:
   if sum(sum(sum(sum(X_gray_chick ==
X_train_lab_temp[check_train])))/100352 > 0.4:
       X_train_l = np.zeros((size_train-1, 224, 224, 1),
dtype=float)
       X_train_l[0:check_train] = X_train_l_temp[0:check_train]
       X_train_l[check_train:] = X_train_l_temp[(check_train+1):]
       X_train_l_temp = X_train_l

       X_train_lab = np.zeros((size_train-1, 224, 224, 2),
```

```
dtype=float)
       X_train_lab[0:check_train] =
X_train_lab_temp[0:check_train]
       X_train_lab[check_train:] = X_train_lab_temp[check_train +
1:]
       X_train_lab_temp = X_train_lab
       size_train = size_train-1
       check_train = check_train - 1
   check_train = check_train + 1

X_test_l = X_test_l/255
X_train_l = X_train_l/255

X_train_l = tf.cast(X_train_l, tf.float32)
X_test_l = tf.cast(X_test_l, tf.float32)

model = Sequential()
inputs = Input((224, 224, 1))
conv1 = Conv2D(32, (3, 3), activation='relu',
padding='same')(inputs)
conv1 = Conv2D(32, (3, 3), activation='relu',
padding='same')(conv1)
pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)
conv2 = Conv2D(64, (3, 3), activation='relu',
padding='same')(pool1)
conv2 = Conv2D(64, (3, 3), activation='relu',
padding='same')(conv2)
pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)
conv3 = Conv2D(128, (3, 3), activation='relu',
padding='same')(pool2)
conv3 = Conv2D(128, (3, 3), activation='relu',
padding='same')(conv3)
pool3 = MaxPooling2D(pool_size=(2, 2))(conv3)
conv4 = Conv2D(256, (3, 3), activation='relu',
padding='same')(pool3)
conv4 = Conv2D(256, (3, 3), activation='relu',
padding='same')(conv4)
pool4 = MaxPooling2D(pool_size=(2, 2))(conv4)
conv5 = Conv2D(512, (3, 3), activation='relu',
padding='same')(pool4)
conv5 = Conv2D(512, (3, 3), activation='relu',
padding='same')(conv5)
```

```
up6 = concatenate([Conv2DTranspose(256, (2, 2), strides=(2, 2),
padding='same')(conv5), conv4], axis=3)
conv6 = Conv2D(256, (3, 3), activation='relu',
padding='same')(up6)
conv6 = Conv2D(256, (3, 3), activation='relu',
padding='same')(conv6)
up7 = concatenate([Conv2DTranspose(128, (2, 2), strides=(2, 2),
padding='same')(conv6), conv3], axis=3)
conv7 = Conv2D(128, (3, 3), activation='relu',
padding='same')(up7)
conv7 = Conv2D(128, (3, 3), activation='relu',
padding='same')(conv7)
up8 = concatenate([Conv2DTranspose(64, (2, 2), strides=(2, 2),
padding='same')(conv7), conv2], axis=3)
conv8 = Conv2D(64, (3, 3), activation='relu', padding='same')(up8)
conv8 = Conv2D(64, (3, 3), activation='relu',
padding='same')(conv8)
up9 = concatenate([Conv2DTranspose(32, (2, 2), strides=(2, 2),
padding='same')(conv8), conv1], axis=3)
conv9 = Conv2D(32, (3, 3), activation='relu', padding='same')(up9)
conv9 = Conv2D(32, (3, 3), activation='relu',
padding='same')(conv9)
conv10 = Conv2D(2, (3, 3), activation='relu',
padding='same')(conv9)

model = Model(inputs=[inputs], outputs=[conv10])
model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
model.summary()

history = model.fit(x=X_train_l, y=X_train_lab,
validation_split=0.2, epochs=400, batch_size=16)
xx = model.predict(X_test_l)
xx = xx.reshape(size_test, 224, 224, 2)

h = history
plt.plot(h.history['loss'])
plt.plot(h.history['val_loss'])
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('Model Loss')
plt.legend(['loss', 'val_loss'])
plt.axis([0, 250, 0, 20000])
```

```
plt.show()
plt.plot(h.history['accuracy'])
plt.plot(h.history['val_accuracy'])
plt.legend(['accuracy', 'val_accuracy'])
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Model Accuracy')
plt.show()
for k in range(size_test):
   # predicted image
   img = np.zeros((224, 224, 3))
   kor = X_test_l[k]*255

   kor = kor[:, :, 0]
   img[:, :, 1:] = xx[k]
   img[:, :, 0] = kor

   img = img.astype('uint8')
   img_ = cv2.cvtColor(img, cv2.COLOR_LAB2RGB)
   plt.imshow(img_)
   plt.show()

   # actual image
   img_truth = np.zeros((224, 224, 3))
   kor_truth = X_test_l[k]*255
   kor_truth = kor_truth[:, :, 0]
   img_truth[:, :, 0] = kor_truth
   img_truth[:, :, 1:] = X_test_lab[k]

   img_truth = img_truth.astype('uint8')
   img_truth_ = cv2.cvtColor(img_truth, cv2.COLOR_LAB2RGB)
   plt.imshow(img_truth_)
   plt.show()
   (score, diff) = compare_ssim(img_truth, img_truth_, full=True,
multichannel=True)
   diff = (diff * 255).astype("uint8")
   print("SSIM: {}".format(score))
   print("PSNR: {}".format(cv2.PSNR(img_truth, img_truth_)))
   m = tf.keras.metrics.Accuracy()
   m.update_state(img_truth, img_truth_)
   print("Accuracy: ", m.result().numpy())
```

Code Block 2: Python code for U-Net [7].

ResNet Python code

```
#Reference: The ResNet Model is taken from the following link
https://github.com/lukemelas/Automatic-Image-Colorization/
# Libraries
import csv
import matplotlib.pyplot as plt
import cv2
import tensorflow as tf
import keras
from skimage.measure import compare_ssim
from tensorflow import keras
from keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten,
BatchNormalization
from keras.layers import Conv2D, MaxPooling2D, InputLayer,
UpSampling2D, Conv2DTranspose, LeakyReLU, AveragePooling2D, \
   GlobalAveragePooling2D, GlobalMaxPooling2D, Input,
RepeatVector, Reshape, concatenate
from keras.utils.np_utils import to_categorical
import matplotlib.pyplot as plt
from keras.callbacks import LearningRateScheduler,
ModelCheckpoint, ReduceLROnPlateau
import numpy as np
from keras import backend as K
from matplotlib.image import imsave
import os
from skimage.color import rgb2gray, gray2rgb, rgb2lab
from skimage.transform import resize
from keras.applications.inception_resnet_v2 import
InceptionResNetV2, preprocess_input
from keras.models import Model
from tensorflow.python.keras.applications.vgg16 import VGG16
from tensorflow.keras.applications.resnet50 import ResNet50
from classification_models.keras import Classifiers


images_gray = np.load("gray_scale.npy")
images_lab = np.load("ab1.npy")
```

```
# gray_scale
size_train = 1000
size_test = 120
size_test_begin = 0
change_pic = 0
X_train_l = images_gray[0+change_pic:change_pic+size_train]
X_train_l = X_train_l.reshape(size_train, 224, 224, 1)
X_test_l = images_gray[size_test_begin +
size_train+change_pic:change_pic+size_train + size_test_begin +
size_test]
X_test_l = X_test_l.reshape(size_test, 224, 224, 1)

# colored
X_train_lab = images_lab[0+change_pic:change_pic+size_train]
X_train_lab = X_train_lab.reshape(size_train, 224, 224, 2)
X_test_lab = images_lab[size_test_begin +
size_train+change_pic:change_pic+size_train + size_test_begin +
size_test]
X_test_lab = X_test_lab.reshape(size_test, 224, 224, 2)

X_gray_check = np.zeros((1,224,224,2), dtype=float)
X_gray_check = X_gray_check.__add__(128)
check_test = 1
X_test_lab_temp = X_test_lab
X_test_l_temp = X_test_l
while check_test < size_test:
   if sum(sum(sum(sum(X_gray_check ==
X_test_lab_temp[check_test])))/100352 > 0.4:
       X_test_l = np.zeros((size_test-1, 224, 224, 1),
dtype=float)
       X_test_l[0:check_test] = X_test_l_temp[0:check_test]
       X_test_l[check_test:] = X_test_l_temp[(check_test+1):]
       X_test_l_temp = X_test_l

       X_test_lab = np.zeros((size_test-1, 224, 224, 2),
dtype=float)
       X_test_lab[0:check_test] = X_test_lab_temp[0:check_test]
       X_test_lab[check_test:] = X_test_lab_temp[check_test + 1:]
       X_test_lab_temp = X_test_lab
       size_test = size_test-1
       check_test = check_test - 1
   check_test = check_test + 1
```

```
check_train = 0
X_train_lab_temp = X_train_lab
X_train_l_temp = X_train_l
while check_train < size_train:
   if sum(sum(sum(sum(X_gray_check ==
X_train_lab_temp[check_train])))/100352 > 0.4:
       X_train_l = np.zeros((size_train-1, 224, 224, 1),
dtype=float)
       X_train_l[0:check_train] = X_train_l_temp[0:check_train]
       X_train_l[check_train:] = X_train_l_temp[(check_train+1):]
       X_train_l_temp = X_train_l

       X_train_lab = np.zeros((size_train-1, 224, 224, 2),
dtype=float)
       X_train_lab[0:check_train] =
X_train_lab_temp[0:check_train]
       X_train_lab[check_train:] = X_train_lab_temp[check_train +
1:]
       X_train_lab_temp = X_train_lab
       size_train = size_train-1
       check_train = check_train - 1
   check_train = check_train + 1

X_test_l = X_test_l/255
X_train_l = X_train_l/255

X_train_l = tf.cast(X_train_l, tf.float32)
X_test_l = tf.cast(X_test_l, tf.float32)


ResNet18, preprocess_input = Classifiers.get('resnet18')
model = Sequential()
model.add(ResNet18(input_shape=(224,224,1),include_top=False,class
es=365))
model.add(Conv2D(64, (3, 3), activation='relu',padding='same'))
model.add(BatchNormalization(batch_size=64))
model.add(UpSampling2D(size=(2,2)))
model.add(Conv2D(64, (3, 3), activation='relu',padding='same'))
model.add(BatchNormalization(batch_size=64))
model.add(UpSampling2D(size=(2,2)))
model.add(Conv2D(32, (3, 3), activation='relu',padding='same'))
```

```
model.add(BatchNormalization(batch_size=32))
model.add(UpSampling2D(size=(2,2)))
model.add(Conv2D(32, (3, 3), activation='relu',padding='same'))
model.add(BatchNormalization(batch_size=32))
model.add(UpSampling2D(size=(2,2)))
model.add(Conv2D(2, (3, 3), activation='relu',padding='same'))
model.add(UpSampling2D(size=(2,2)))


model.compile(optimizer='adam', loss='mse', metrics=['accuracy'])
model.summary()
history = model.fit(x=X_train_l, y=X_train_lab,
validation_split=0.2, epochs=500, batch_size=32)

xx = model.predict(X_test_l)
xx = xx.reshape(size_test, 224, 224, 2)

h = history
plt.plot(h.history['loss'])
plt.plot(h.history['val_loss'])
plt.xlabel('epoch')
plt.ylabel('loss')
plt.title('Model Loss')
plt.legend(['loss', 'val_loss'])
plt.axis([0, 250, 0, 20000])
plt.show()
plt.plot(h.history['accuracy'])
plt.plot(h.history['val_accuracy'])
plt.legend(['accuracy', 'val_accuracy'])
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.title('Model Accuracy')
plt.show()
for k in range(size_test):
   # predicted image
   img = np.zeros((224, 224, 3))
   kor = X_test_l[k]

   kor = kor[:, :, 0]
   img[:, :, 1:] = xx[k]
   img[:, :, 0] = kor
```

```
    img = img.astype('uint8')
    img_ = cv2.cvtColor(img, cv2.COLOR_LAB2RGB)
    plt.imshow(img_)
    plt.show()

    # actual image
    img_truth = np.zeros((224, 224, 3))
    kor_truth = X_test_l[k]
    kor_truth = kor_truth[:, :, 0]
    img_truth[:, :, 0] = kor_truth
    img_truth[:, :, 1:] = X_test_lab[k]

    img_truth = img_truth.astype('uint8')
    img_truth_ = cv2.cvtColor(img_truth, cv2.COLOR_LAB2RGB)
    plt.imshow(img_truth_)
    plt.show()
    (score, diff) = compare_ssim(img_truth, img_truth_, full=True,
multichannel=True)
    diff = (diff * 255).astype("uint8")
    print("SSIM: {}".format(score))
    print("PSNR: {}".format(cv2.PSNR(img_truth, img_truth_)))
```

Code Block 3: Python code for ResNet [5].

VIII. References

[1] "Black and white image colorization with OpenCV and Deep Learning," *PyImageSearch*, Feb. 25, 2019. https://www.pyimagesearch.com/2019/02/25/black-and-white-image-colorization-with-opencv-and-deep-learning/ (accessed Nov. 15, 2020).

[2] "Image Colorization." https://kaggle.com/shravankumar9892/image-colorization (accessed Nov. 15, 2020).

[3] "Convolutional Neural Network." https://www.mathworks.com/discovery/convolutional-neural-network-matlab.html (accessed Dec. 18, 2020).

[4] A. Wong, "Traffic Sign Classification," Medium, 31-Dec-2018. [Online]. Available: https://medium.com/hackernoon/traffic-sign-classification-6e7113d9c4d5. [Accessed: 19-Dec-2020].

[5] Lukemelas. (n.d.). Lukemelas/Automatic-Image-Colorization. Retrieved December 19, 2020, from https://github.com/lukemelas/Automatic-Image-Colorization/

[6] H. Lamba, "Understanding Semantic Segmentation with UNET," *Medium*, Feb. 17, 2019. https://towardsdatascience.com/understanding-semantic-segmentation-with-unet-6be4f42d4b47 (accessed Dec. 18, 2020).

[7] Rahuldshetty. (2019, October 23). Image Colorization with UNET-Auto Encoders. Retrieved December 19, 2020, from https://www.kaggle.com/rahuldshetty/image-colorization-with-unet-auto-encoders/notebook

[8] A. KARAZOR, "Görüntü Kalite Metrikleri Nedir? MATLAB İle Görüntü Kalite Metrikleri Hesaplama Arayüzü," *Medium*, Oct. 24, 2019. https://medium.com/@ahmetkarazor/g%C3%B6r%C3%BCnt%C3%BC-kalite-metrikleri-nedir-matlab-i%CC%87le-g%C3%B6r%C3%BCnt%C3%BC-kalite-metrikleri-hesaplama-aray%C3%BCz%C3%BC-b51596992477 (accessed Dec. 19, 2020).