Course Project Part II
Neural Networks & Computer Graphics

Akshay Sharma (akshaysharma@ufl.edu)
Malvika Ranjitsinh Jadhav (jadhav.m@ufl.edu)
Tanvi Jain (tjain@ufl.edu)

1 Overview

This project consists of two main parts.The first part focuses on building a model to perform automatic colorization of input black and white images. The second part consists of performing strategic and efficient transfer learning exercise of the network model built in first part with an aim to achieve best possible performance on the new dataset.

2 Instruction To Run The Code

There are few code files as part of this project. In order to run the files, unzip folders in *Dataset* folder and also unzip *ncd_predicted_images.zip* (This file has the NCD predicted images on CPU) *predicted_images.zip* (This file has the face_images predicted on GPU), *NCD_predictions.zip* (This file has the NCD predicted images on GPU), *cpu_predicted_images.zip* (This file has the face_images predicted on CPU)

1. *Part2.ipynb* contains the code for preprocessing the dataset which includes, train-test split, augmentation and conversion on images to L*a*b color space.
2. *mean_chrominance.ipynb*: Regressor model is implemented in this file which returns the mean chrominance.
3. *colorizer_model_cpu.ipynb*: Colorizer for face_images dataset as well as the transfer learning part for NCD dataset is coded in this file. This will create the predicted_images and ncd_predicted_images as well as the test scores.*colorizer_model.ipynb* is the file with similar functionality and also has the GPU computing code.
4. *NCD_transfer learning.ipynb* has the code to for transfer learning on NCD dataset.
5. *requirement.txt* enlist the dependencies to run this project.

3 System Design:

We started our project by exploring the Georgia Tech Face Dataset. We then performed the following preprocessing steps to get the dataset ready to be in appropriate input format for our colorization model. In order to read the dataset, we have used multiple methods in order to explore various libraries. Data loading using dataloader class is implemented in regression and colorizer code.

3.1 Preprocessing:

Preprocessing the data has been divided in overall three sections.

1. Split the folder containing original face_images data in a train and test folders with a seed value and 9:1 ratio. For this we have used library *splifolders*. We did this to get the training data set that can be augmented in the next step.
2. To augment the training data to 10 times the original count we have created *torchvision.transforms* methods for horizontal flips, vertical flips, horizontal & vertical flips, random crop and also added a scaler between [0.6-1] to the RGB values.

(a) *horizontalFlip*: Transform the image array to PIL image and then call **RandomHorizontalFlip** with a probability parameter: p = 1 to flip all the images.

(b) *verticalFlip*: Transform the image array to PIL image and then call **RandomVerticallFlip** with a probability parameter: p = 1 to flip all the images.

(c) *twoFlips*: Transform the image array to PIL image and then call **RandomHorizontalFlip** followed by **RandomVerticallFlip** on the same image with a probability parameter: p = 1.

(d) *randomCrop*: Transform the image array to PIL image and then call **RandomResizedCrop** with a crop image size of (64,64). Thsi will cop teh orginal image randomly and then resize it to the original dimensions.

(e) To add a scaler to the RGB image, we have randomly opted a number between [0.6, 1] with 1 floating point. Then added this scaler to the normalized RGB image fetched using **matplotlib.pyplot**. Then in order to use and save this image further, converted this image array to a PIL image using the **Image** method of PIL library.

We have created a function *augment_data* that calls the above mentioned augmentation options, and saved new images in a folder *Dataset/augment_data*. Images stored has the original image name followed by a _¡number¿ where number is between [0,9].

3. Next, to convert the augmented images and test images to L*a*b color space, we have created a function *pre_process* which reads the files from input folder using glob.glob and then loop over the files. within loop, call cv2.imread(¡file_name¿) to read the image in BGR format folowed by cv2.resize(¡image¿, (128,128)) to resize the read image. Next is to call cv2.cvtColor(¡image¿, cv2.COLOR_BGR2LAB) to convert the BGR image to L*a*b color space. Then call cv2.split(¡image¿) to split the L*a*b color space image to L, a, and b. Save these images in their respective folders. We have also created a folder containing all images (L, a, b, input, L*a*b color space image).

   This function is called for images in folder *Dataset/augment_data* and processed images are stored in

   Dataset/l_data, *Dataset/a_data*, *Dataset/b_data*, *Dataset/lab_data*, *Dataset/all_data*.

   This function is also called for images in test folder i.e., *Dataset/face_images/test/face_images* and corresponding processed images are stored in folders

   Dataset/l_data/test, *Dataset/a_data/test*, *Dataset/b_data/test*, *Dataset/lab_data/test*, *Dataset/all_data/test*

3.2 Model Building:

3.2.1 Input Creation:

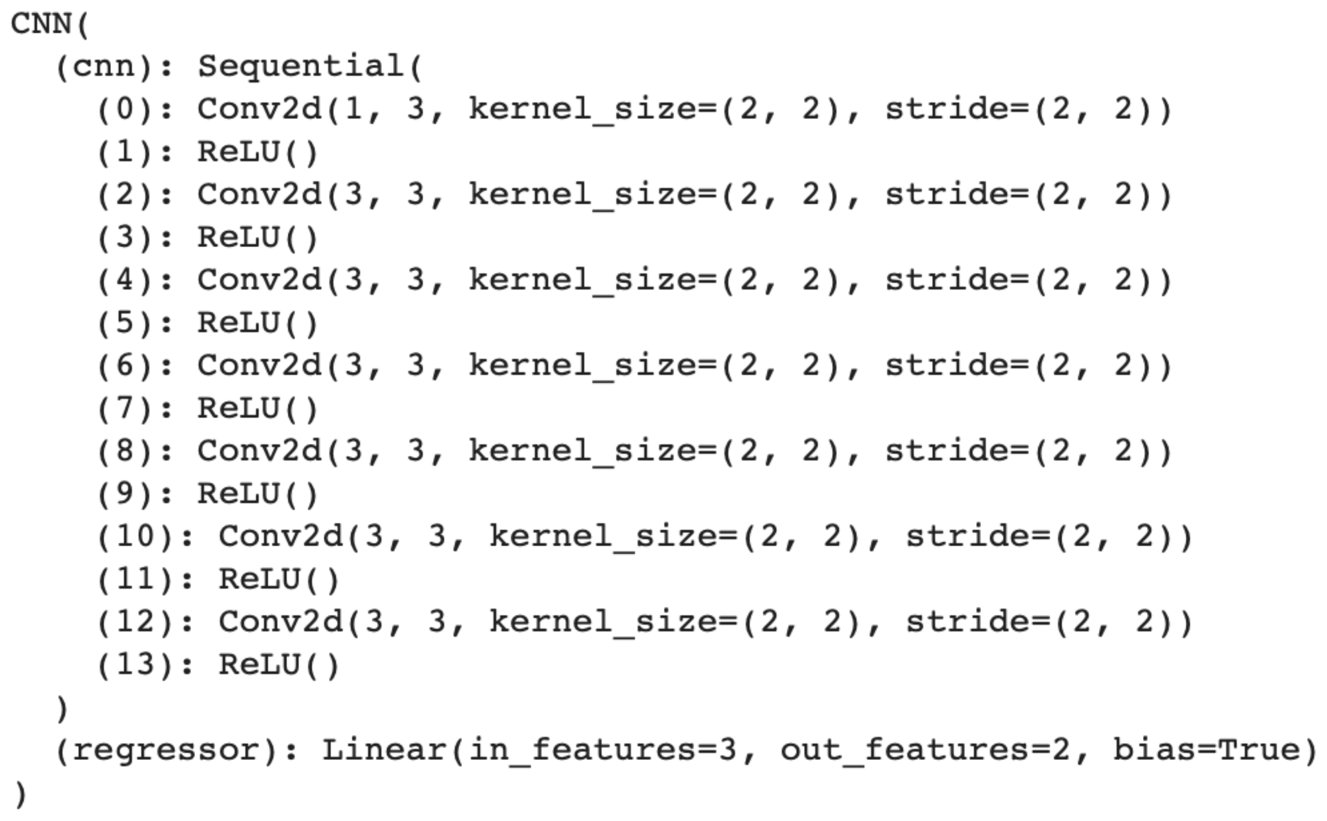
The input to the regressor model is the L channel tensor extracted from each image of original dataset(face dataset). We first normalize the values of L, a and b channels before using deep learning models. The normalized L channel tensor is input for the CNN and the list [a, b] are used as target values. The list [a, b] is constructed by computing mean pixel value for a and b channels of each sample from input dataset.

3.2.2 Simple Regressor Model:

1. Hidden layers: For the hidden layers we have used spatial convolution layers. Every layer convolves to produce a resulting tensor (feature map) of lower dimension. We have written 7 such convolution layers in the regression model. The dimensions of input image reduce in the following sequence:

layer 1: 128 x 128
layer 2: 64 x 64
layer 3: 32 x 32
layer 4: 16 x 16
layer 5: 8 x 8
layer 6: 4 x 4
layer 7: 2 x 2

Table 1: Dimension of input image tensor at each hidden layer



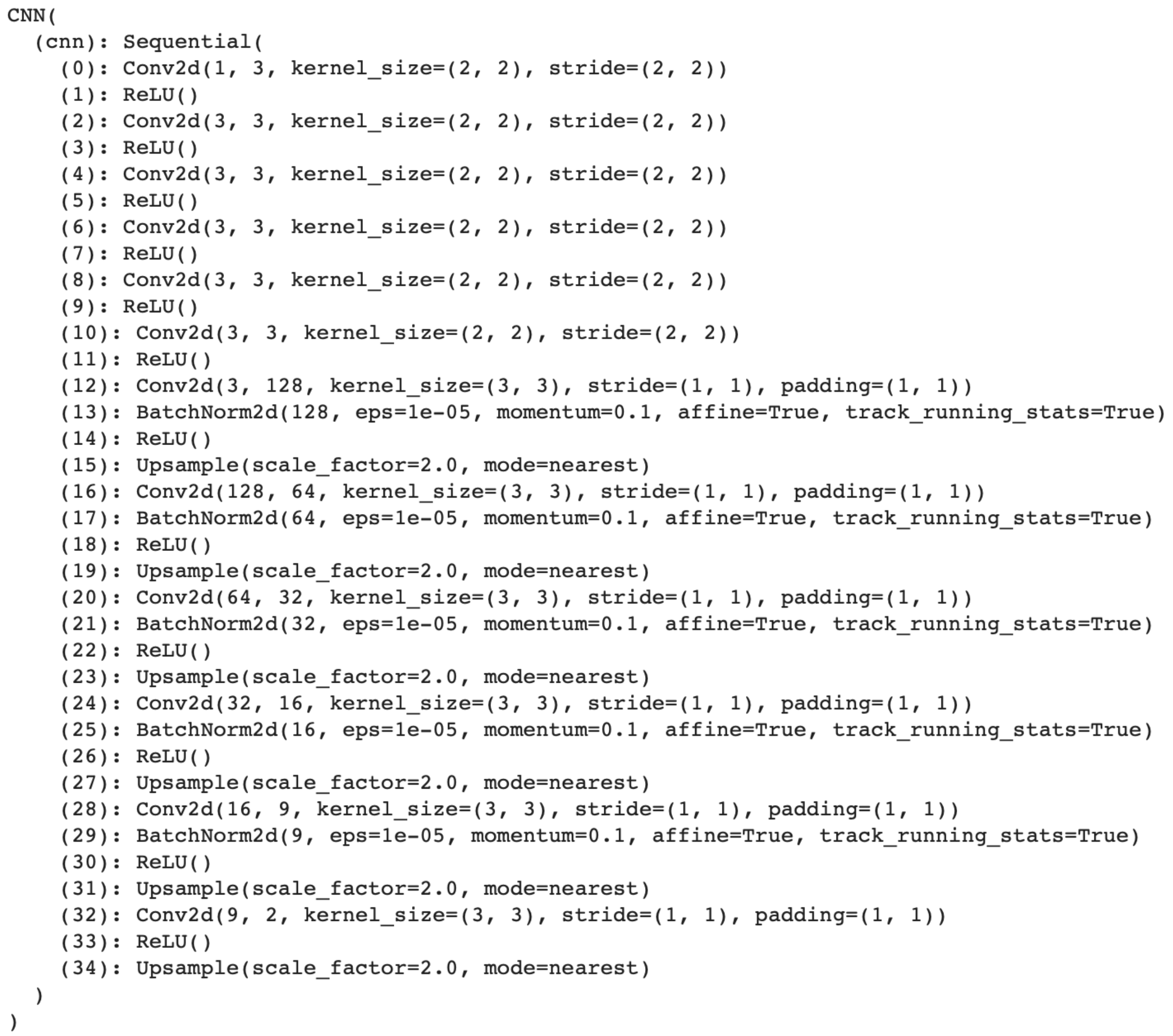
```
CNN(
  (cnn): Sequential(
    (0): Conv2d(1, 3, kernel_size=(2, 2), stride=(2, 2))
    (1): ReLU()
    (2): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (5): ReLU()
    (6): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (7): ReLU()
    (8): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (9): ReLU()
    (10): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (11): ReLU()
    (12): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (13): ReLU()
  )
  (regressor): Linear(in_features=3, out_features=2, bias=True)
)
```

Figure 1: Architecture for getting mean chrominance

2. Output layer: Since we get a 2 x 2 output at the end of last convolutional layer we flatten the tensor and add a linear layer to map it to a 2 x 1 output. This is because we want only 2 scalar chrominance values as the output of our regressor model.

3.2.3 Colorization model:

Building on the simple regressor we have previously constructed we now add few more layers to transform the regressor into a colorization model. We now need to construct a model that can output the entire a and b channels as opposed to just the mean chrominance value in the former case. The architecture of our colorization model is as follows:



```
CNN(
  (cnn): Sequential(
    (0): Conv2d(1, 3, kernel_size=(2, 2), stride=(2, 2))
    (1): ReLU()
    (2): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (3): ReLU()
    (4): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (5): ReLU()
    (6): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (7): ReLU()
    (8): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (9): ReLU()
    (10): Conv2d(3, 3, kernel_size=(2, 2), stride=(2, 2))
    (11): ReLU()
    (12): Conv2d(3, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU()
    (15): Upsample(scale_factor=2.0, mode=nearest)
    (16): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (18): ReLU()
    (19): Upsample(scale_factor=2.0, mode=nearest)
    (20): Conv2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (21): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (22): ReLU()
    (23): Upsample(scale_factor=2.0, mode=nearest)
    (24): Conv2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (26): ReLU()
    (27): Upsample(scale_factor=2.0, mode=nearest)
    (28): Conv2d(16, 9, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): BatchNorm2d(9, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (30): ReLU()
    (31): Upsample(scale_factor=2.0, mode=nearest)
    (32): Conv2d(9, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU()
    (34): Upsample(scale_factor=2.0, mode=nearest)
  )
)
```

Figure 2: Architecture for colorizer

For our final architecture we got average means square error values as follows:

Train dataset:0.0055
Test dataset: 0.0078

Table 2: MSE values

The plot for average MSE per epoch for our architecture during the training stage appeared as

3.3 Transfer Learning:

For this part of the project we are analyzing the transferability of our model to images from another domain. For this experiment we are using the NCD dataset to run aor pre-trained model on face_images dataset.

3.3.1 Evaluation Methods:

To evaluate the performance of tranfer learning, we have used Structural Similarity Index (SSIM).

4 Results

4.0.1 Preprocessing Results

Augment images example are in Figure 3

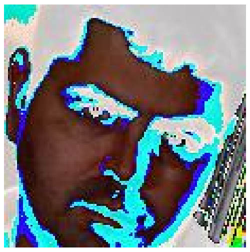
(a) Original

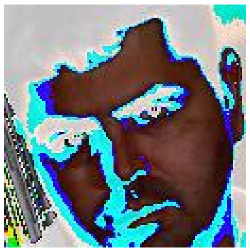
(b) Horizontal flip

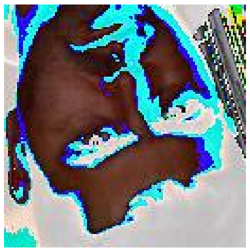
(c) Vertical Flip

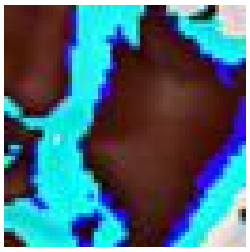
(d) Random Crop

(e) H & V Flip


(f) RGB Scaled


(g) Horizontal flip


(h) Vertical Flip


(i) Random Crop

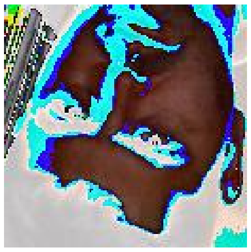
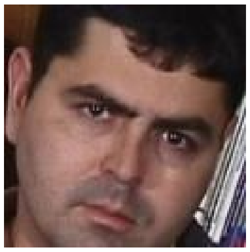
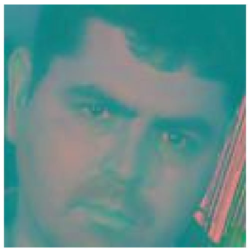
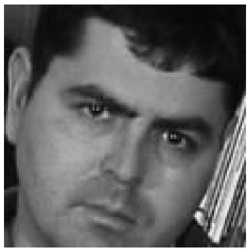

(j) H & V Flip

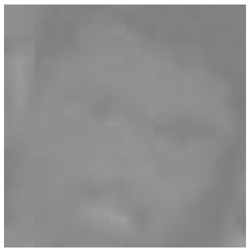
Figure 3: Augmented Data

Example of Original with L*a*b, L, a and b channel images is in Figure 4


(a) Original


(b) L*a*b


(c) L


(d) a

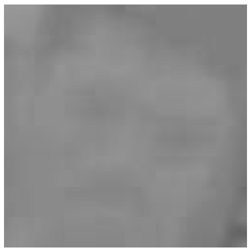

(e) b

Figure 4: Processed images in RGB and L*a*b color space

Augmented images are stored in folder: *./Dataset/augment_data*
L*a*b train images are stored in folder: *./Dataset/lab_data*
L train images are stored in folder: *./Dataset/l_data*
a train images are stored in folder: *./Dataset/a_data*
b train images are stored in folder: *./Dataset/b_data*
Test dataset is stored in folder: *./Dataset/face_images/test/face_images*
L*a*b test images are stored in folder: *./Dataset/lab_data*
L train images are stored in folder: *./Dataset/l_data/test*
a test images are stored in folder: *./Dataset/a_data/test*
b test images are stored in folder: *./Dataset/b_data/test*

4.0.2 Simple regressor:

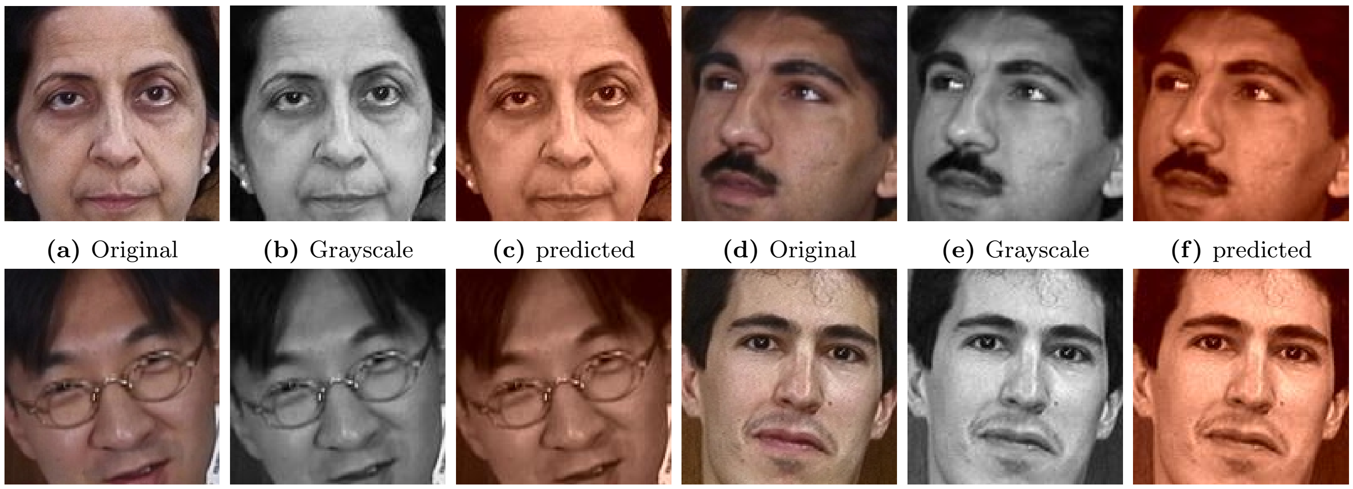
After constructing our simple regression model our average MSE value for the entire training set was 0.0117. *chrominance_model.pt* is the model for regressor returning the mean chrominance. Few of our predicted channel results are enclosed in the table below:

achannel	bchannel	predicted achannel	predicted bchannel	MSE
0.150	0.7940	0.6719	0.7809	0.00113
0.618	0.7948	0.6671	0.7793	0.00027

Table 3: MSE values

4.0.3 Colorization model:

Figure 8 contains few examples of the colorizer output on the test dataset for face_images dataset. Figure 6 displays the average MSE for epoch values for 20 epochs. *predicted_images* folder in final zip file contains the outcome of colorizer on the grayscale images for test dataset. *colorizer_cpu_model.pt* is the model for colorizer with 10 epochs. *colorizer_250.pt* is the model for colorizer with 250 epochs ran on GPU.



(a) Original **(b)** Grayscale **(c)** predicted **(d)** Original **(e)** Grayscale **(f)** predicted

(g) Original **(h)** Grayscale **(i)** predicted **(j)** Original **(k)** Grayscale **(l)** predicted

Figure 5: Comparative examples for Original, input grayscale and the predicted images from test dataset

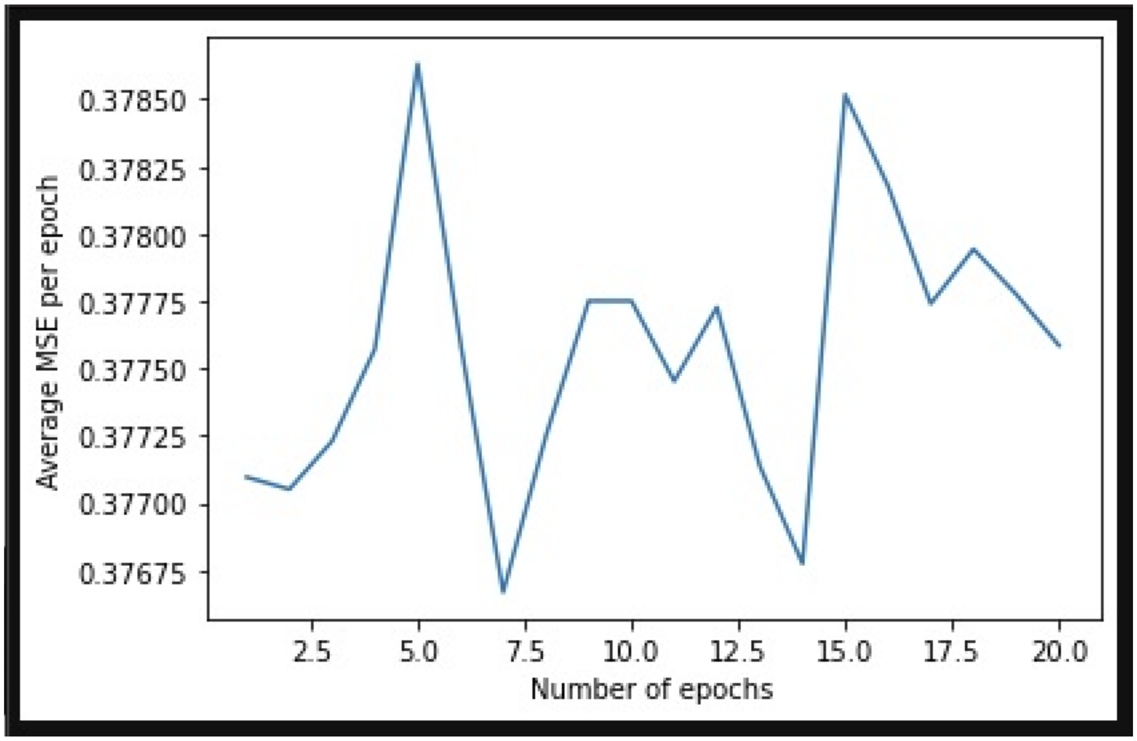
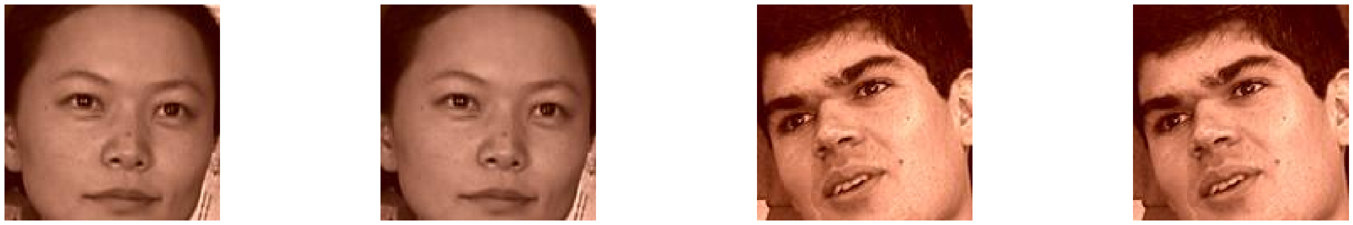


Figure 6: Average MSE for colorizer around 20 epochs

- **GPU Computing**: Training time of the colorizer model(with 10 epochs) before using GPU computing was 487 seconds. After the use of NVIDIA GeForce RTX 2080 Ti GPU for training the train time was reduced to 196 seconds. Followed by which we trained the model for 250 epochs using the GPU, for which the training time was 6111 seconds.The difference in result of prediction for both the models is displayed in the Figure 7. There are artifacts which are visible in CPU trained model for 10 epochs that are less visible in colorizer model trained with 250 epochs.



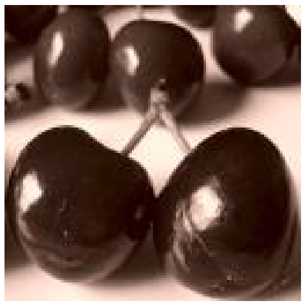
(a) Original **(b)** Grayscale **(c)** predicted **(d)** predicted

Figure 7: Comparative examples for CPU and GPU predicted images

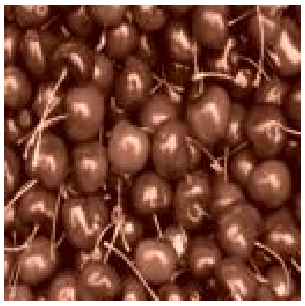
4.0.4 Transfer Learning:

ncd_predicted_images folder in final zip file contains the outcome of transfer learning on the grayscale images of NCD dataset. We have tested the performance of our colorizer model by performing transfer learning. For quite a few samples we were able to achieve SSIM value as high as 0.6-0.7. Overall our model shows decent potential for transferability. *ssim_.txt* is the file that contains SSIM values for colourful NCD images and predicted images by our model.

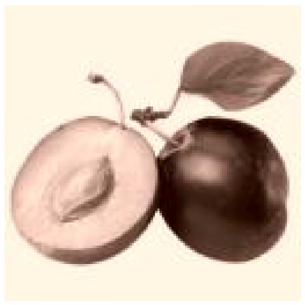
(a) Grayscale

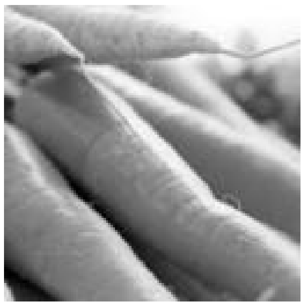

(b) Predicted

(c) Grayscale


(d) predicted

(e) Grayscale


(f) predicted


(g) Grayscale

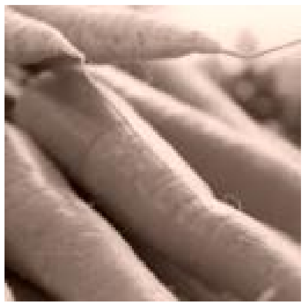

(h) Predicted

Figure 8: Comparative examples for grayscale and the predicted images from NCD dataset

5 Conclusion

We have successfully built a colorizer model and performed transfer learning.Our predicted images have a positive ssim value and our model is able to predict the color components of unacquainted domains to an decent extent. Our model lacks in generalizing all color channels with equivalent weigtage but training on a variety of input images may help eradicate the issue.

References

[1] https://mmuratarat.github.io/2020-01-25/multilabel_classification_metrics.

[2] https://stackoverflow.com/questions/51677788/data-augmentation-in-pytorch.

[3] https://www.javatpoint.com/machine-learning-support-vector-machine-algorithm.