CAP5404 Deep Learning for Computer Graphics
Dr. Corey Toler-Franklin
Course Project Part II Neural Networks & Computer Graphics

Sai Nikhil Dondapati - 22286439

1) Data Processing and Data Augmentation

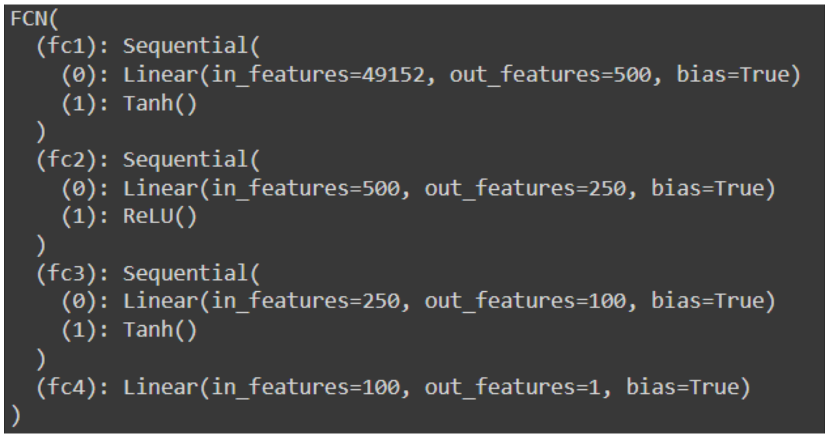
- The original dataset contains 750 colored images.
- The augmentations performed were horizontal flipping, cropping and scaling.
- There are 950 images after data augmentation.
- The data is then transformed from BGR to LAB.
- LAB images were split into L*, a* and b* channels. The L* channel is also called the lightness channel whereas a* and b* channels correspond to colors red, green, blue and yellow. The L* channel is used as input for the model.

2) Regressor

The regressor takes the L* channel as an input and predicts the mean of a* and b* channels. This was implemented using a fully connected network (FCN) and a Convolutional neural network (CNN).

Fully Connected Network:
A fully connected neural network consists of a series of fully connected layers where every neuron in a layer is connected to every other neuron in the other layer.
The FCN had three fully connected layers and one output layer.

Best FCN Architecture:

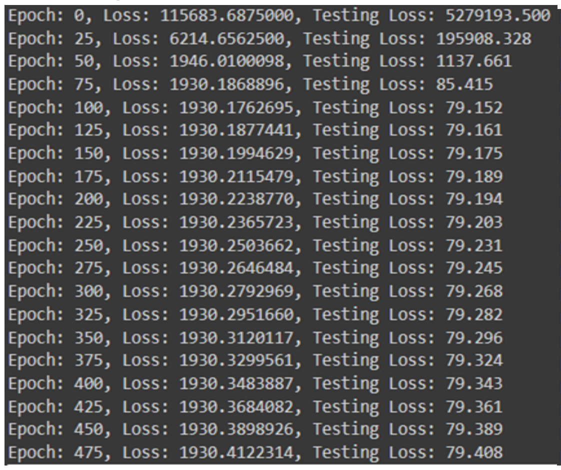


```
FCN(
  (fc1): Sequential(
    (0): Linear(in_features=49152, out_features=500, bias=True)
    (1): Tanh()
  )
  (fc2): Sequential(
    (0): Linear(in_features=500, out_features=250, bias=True)
    (1): ReLU()
  )
  (fc3): Sequential(
    (0): Linear(in_features=250, out_features=100, bias=True)
    (1): Tanh()
  )
  (fc4): Linear(in_features=100, out_features=1, bias=True)
)
```

After training the model, we predict the mean of a* and b*, and plot Mean Squared Error (MSE) vs the number of epochs.

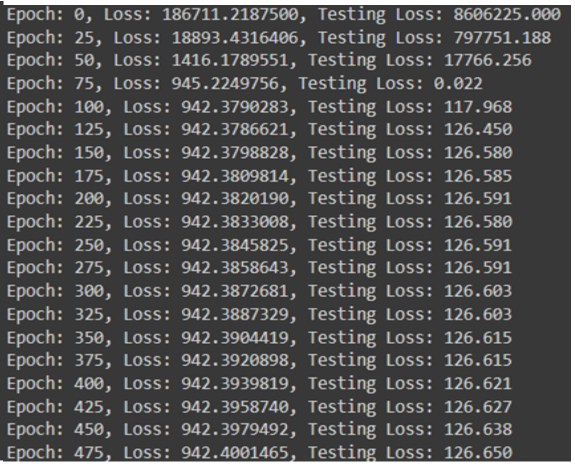
FCN Results:

Predicting mean a*



```
Epoch: 0, Loss: 115683.6875000, Testing Loss: 5279193.500
Epoch: 25, Loss: 6214.6562500, Testing Loss: 195908.328
Epoch: 50, Loss: 1946.0100098, Testing Loss: 1137.661
Epoch: 75, Loss: 1930.1868896, Testing Loss: 85.415
Epoch: 100, Loss: 1930.1762695, Testing Loss: 79.152
Epoch: 125, Loss: 1930.1877441, Testing Loss: 79.161
Epoch: 150, Loss: 1930.1994629, Testing Loss: 79.175
Epoch: 175, Loss: 1930.2115479, Testing Loss: 79.189
Epoch: 200, Loss: 1930.2238770, Testing Loss: 79.194
Epoch: 225, Loss: 1930.2365723, Testing Loss: 79.203
Epoch: 250, Loss: 1930.2503662, Testing Loss: 79.231
Epoch: 275, Loss: 1930.2646484, Testing Loss: 79.245
Epoch: 300, Loss: 1930.2792969, Testing Loss: 79.268
Epoch: 325, Loss: 1930.2951660, Testing Loss: 79.282
Epoch: 350, Loss: 1930.3120117, Testing Loss: 79.296
Epoch: 375, Loss: 1930.3299561, Testing Loss: 79.324
Epoch: 400, Loss: 1930.3483887, Testing Loss: 79.343
Epoch: 425, Loss: 1930.3684082, Testing Loss: 79.361
Epoch: 450, Loss: 1930.3898926, Testing Loss: 79.389
Epoch: 475, Loss: 1930.4122314, Testing Loss: 79.408
```

Predicting mean b*



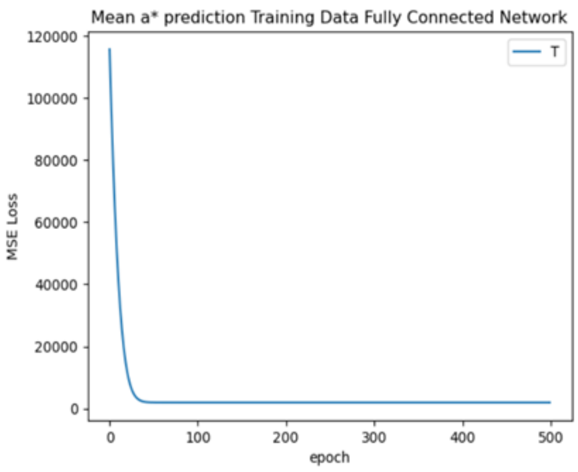
```
Epoch: 0, Loss: 186711.2187500, Testing Loss: 8606225.000
Epoch: 25, Loss: 18893.4316406, Testing Loss: 797751.188
Epoch: 50, Loss: 1416.1789551, Testing Loss: 17766.256
Epoch: 75, Loss: 945.2249756, Testing Loss: 0.022
Epoch: 100, Loss: 942.3790283, Testing Loss: 117.968
Epoch: 125, Loss: 942.3786621, Testing Loss: 126.450
Epoch: 150, Loss: 942.3798828, Testing Loss: 126.580
Epoch: 175, Loss: 942.3809814, Testing Loss: 126.585
Epoch: 200, Loss: 942.3820190, Testing Loss: 126.591
Epoch: 225, Loss: 942.3833008, Testing Loss: 126.580
Epoch: 250, Loss: 942.3845825, Testing Loss: 126.591
Epoch: 275, Loss: 942.3858643, Testing Loss: 126.591
Epoch: 300, Loss: 942.3872681, Testing Loss: 126.603
Epoch: 325, Loss: 942.3887329, Testing Loss: 126.603
Epoch: 350, Loss: 942.3904419, Testing Loss: 126.615
Epoch: 375, Loss: 942.3920898, Testing Loss: 126.615
Epoch: 400, Loss: 942.3939819, Testing Loss: 126.621
Epoch: 425, Loss: 942.3958740, Testing Loss: 126.627
Epoch: 450, Loss: 942.3979492, Testing Loss: 126.638
Epoch: 475, Loss: 942.4001465, Testing Loss: 126.650
```

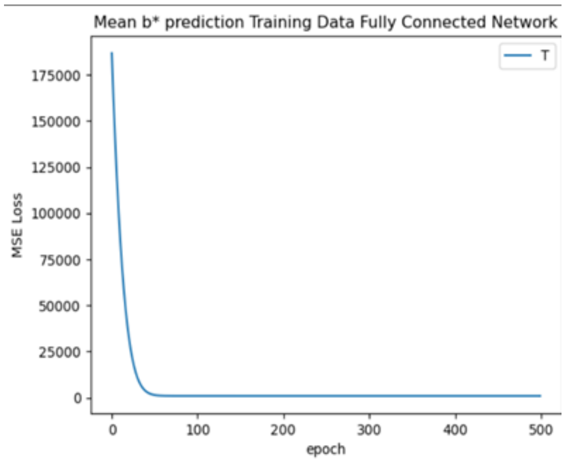
Best MSE achieved on Training set for mean a* prediction: 1930.412
Best MSE achieved on Test set for mean a* prediction: 79.408

Best MSE achieved on Training set for mean b* prediction: 942.4
Best MSE achieved on Test set for mean b* prediction: 126.65

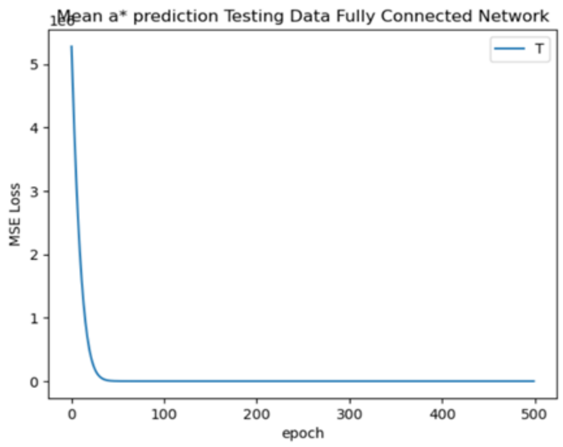
PLOTS:

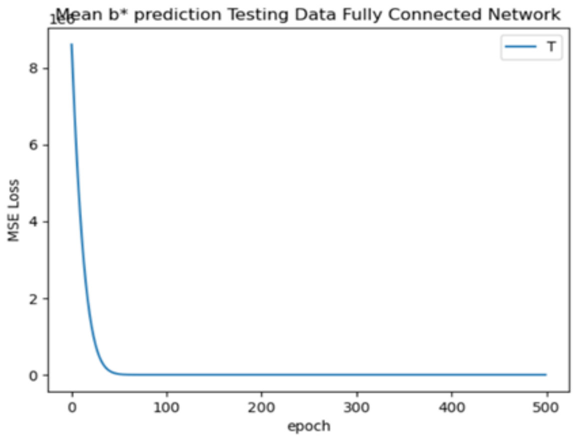
Training MSE:





Testing MSE:





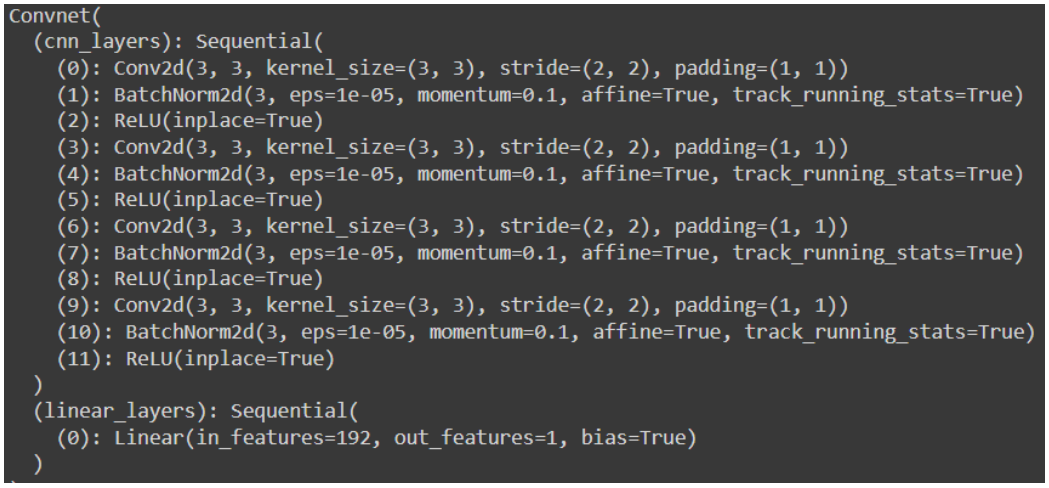
Convolutional Neural Network:

A Convolutional Neural Network CNN) takes an input image, assigns weights to different aspects of the image that can differentiate it from other images.

The first step is to reshape the dataset from (128*128*3) to (3*128*128) for training CNNs.

The CNN used had three convolutional layers with ReLU Activation.

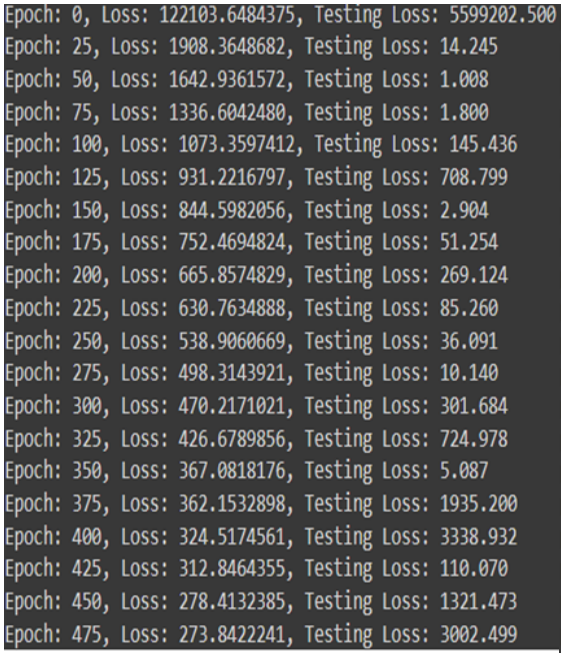
Best Architecture:



```
Convnet(
  (cnn_layers): Sequential(
    (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (1): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(3, 3, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (4): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): Conv2d(3, 3, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): Conv2d(3, 3, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
    (10): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
  )
  (linear_layers): Sequential(
    (0): Linear(in_features=192, out_features=1, bias=True)
  )
```

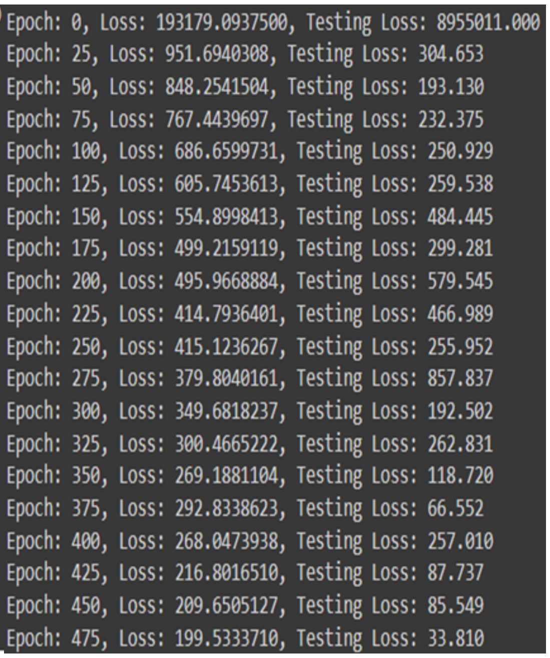
CNN Results:

Predicting mean a*



```
Epoch: 0, Loss: 122103.6484375, Testing Loss: 5599202.500
Epoch: 25, Loss: 1908.3648682, Testing Loss: 14.245
Epoch: 50, Loss: 1642.9361572, Testing Loss: 1.008
Epoch: 75, Loss: 1336.6042480, Testing Loss: 1.800
Epoch: 100, Loss: 1073.3597412, Testing Loss: 145.436
Epoch: 125, Loss: 931.2216797, Testing Loss: 708.799
Epoch: 150, Loss: 844.5982056, Testing Loss: 2.904
Epoch: 175, Loss: 752.4694824, Testing Loss: 51.254
Epoch: 200, Loss: 665.8574829, Testing Loss: 269.124
Epoch: 225, Loss: 630.7634888, Testing Loss: 85.260
Epoch: 250, Loss: 538.9060669, Testing Loss: 36.091
Epoch: 275, Loss: 498.3143921, Testing Loss: 10.140
Epoch: 300, Loss: 470.2171021, Testing Loss: 301.684
Epoch: 325, Loss: 426.6789856, Testing Loss: 724.978
Epoch: 350, Loss: 367.0818176, Testing Loss: 5.087
Epoch: 375, Loss: 362.1532898, Testing Loss: 1935.200
Epoch: 400, Loss: 324.5174561, Testing Loss: 3338.932
Epoch: 425, Loss: 312.8464355, Testing Loss: 110.070
Epoch: 450, Loss: 278.4132385, Testing Loss: 1321.473
Epoch: 475, Loss: 273.8422241, Testing Loss: 3002.499
```

Predicting mean b*



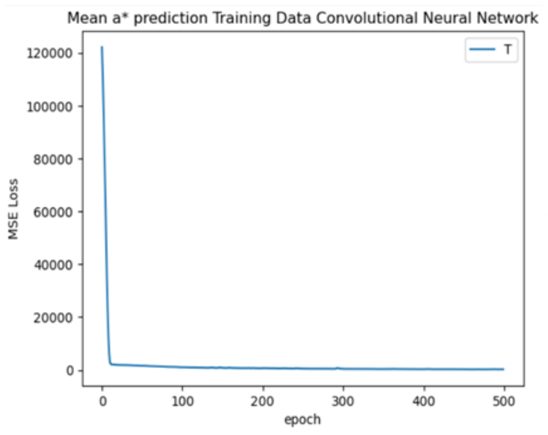
```
Epoch: 0, Loss: 193179.0937500, Testing Loss: 8955011.000
Epoch: 25, Loss: 951.6940308, Testing Loss: 304.653
Epoch: 50, Loss: 848.2541504, Testing Loss: 193.130
Epoch: 75, Loss: 767.4439697, Testing Loss: 232.375
Epoch: 100, Loss: 686.6599731, Testing Loss: 250.929
Epoch: 125, Loss: 605.7453613, Testing Loss: 259.538
Epoch: 150, Loss: 554.8998413, Testing Loss: 484.445
Epoch: 175, Loss: 499.2159119, Testing Loss: 299.281
Epoch: 200, Loss: 495.9668884, Testing Loss: 579.545
Epoch: 225, Loss: 414.7936401, Testing Loss: 466.989
Epoch: 250, Loss: 415.1236267, Testing Loss: 255.952
Epoch: 275, Loss: 379.8040161, Testing Loss: 857.837
Epoch: 300, Loss: 349.6818237, Testing Loss: 192.502
Epoch: 325, Loss: 300.4665222, Testing Loss: 262.831
Epoch: 350, Loss: 269.1881104, Testing Loss: 118.720
Epoch: 375, Loss: 292.8338623, Testing Loss: 66.552
Epoch: 400, Loss: 268.0473938, Testing Loss: 257.010
Epoch: 425, Loss: 216.8016510, Testing Loss: 87.737
Epoch: 450, Loss: 209.6505127, Testing Loss: 85.549
Epoch: 475, Loss: 199.5333710, Testing Loss: 33.810
```

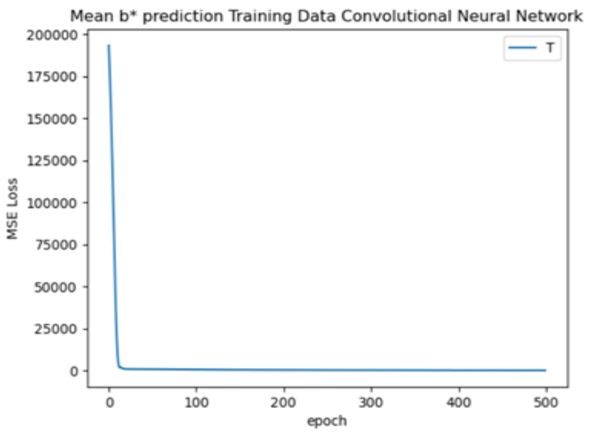
Best MSE achieved on Training set for mean a* prediction: 273.8
Best MSE achieved on Test set for mean a* prediction: 2.9

Best MSE achieved on Training set for mean b* prediction: 199.5
Best MSE achieved on Test set for mean b* prediction: 33.81

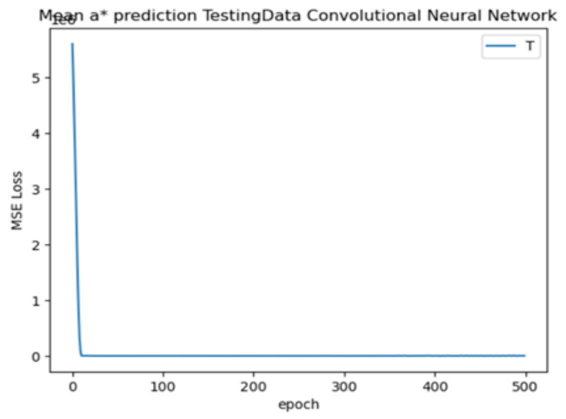
PLOTS:

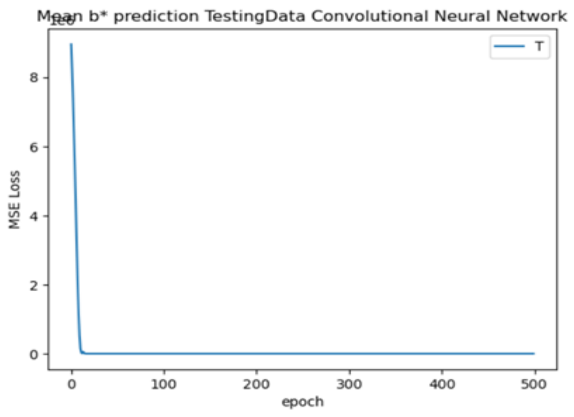
Training MSE:





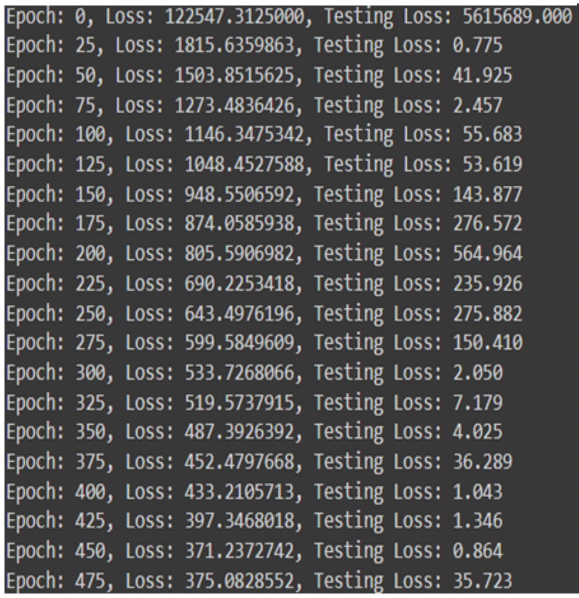
Testing MSE:



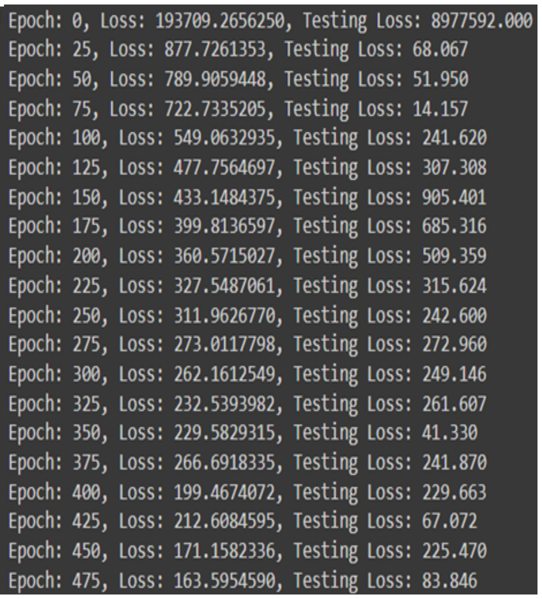


CNN Results after input scaling:

Predicting a*:


```
Epoch: 0, Loss: 122547.3125000, Testing Loss: 5615689.000
Epoch: 25, Loss: 1815.6359863, Testing Loss: 0.775
Epoch: 50, Loss: 1503.8515625, Testing Loss: 41.925
Epoch: 75, Loss: 1273.4836426, Testing Loss: 2.457
Epoch: 100, Loss: 1146.3475342, Testing Loss: 55.683
Epoch: 125, Loss: 1048.4527588, Testing Loss: 53.619
Epoch: 150, Loss: 948.5506592, Testing Loss: 143.877
Epoch: 175, Loss: 874.0585938, Testing Loss: 276.572
Epoch: 200, Loss: 805.5906982, Testing Loss: 564.964
Epoch: 225, Loss: 690.2253418, Testing Loss: 235.926
Epoch: 250, Loss: 643.4976196, Testing Loss: 275.882
Epoch: 275, Loss: 599.5849609, Testing Loss: 150.410
Epoch: 300, Loss: 533.7268066, Testing Loss: 2.050
Epoch: 325, Loss: 519.5737915, Testing Loss: 7.179
Epoch: 350, Loss: 487.3926392, Testing Loss: 4.025
Epoch: 375, Loss: 452.4797668, Testing Loss: 36.289
Epoch: 400, Loss: 433.2105713, Testing Loss: 1.043
Epoch: 425, Loss: 397.3468018, Testing Loss: 1.346
Epoch: 450, Loss: 371.2372742, Testing Loss: 0.864
Epoch: 475, Loss: 375.0828552, Testing Loss: 35.723
```

Predicting b*:

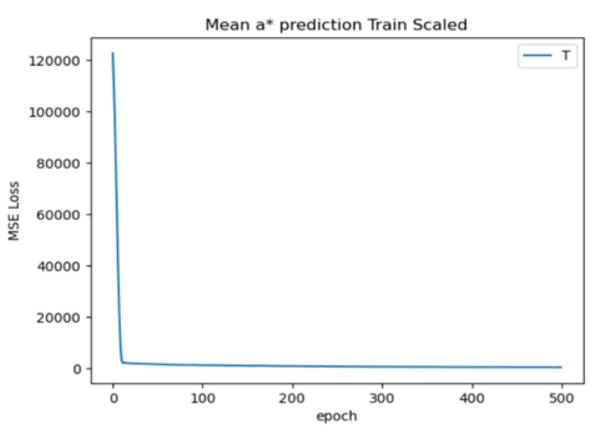

```
Epoch: 0, Loss: 193709.2656250, Testing Loss: 8977592.000
Epoch: 25, Loss: 877.7261353, Testing Loss: 68.067
Epoch: 50, Loss: 789.9059448, Testing Loss: 51.950
Epoch: 75, Loss: 722.7335205, Testing Loss: 14.157
Epoch: 100, Loss: 549.0632935, Testing Loss: 241.620
Epoch: 125, Loss: 477.7564697, Testing Loss: 307.308
Epoch: 150, Loss: 433.1484375, Testing Loss: 905.401
Epoch: 175, Loss: 399.8136597, Testing Loss: 685.316
Epoch: 200, Loss: 360.5715027, Testing Loss: 509.359
Epoch: 225, Loss: 327.5487061, Testing Loss: 315.624
Epoch: 250, Loss: 311.9626770, Testing Loss: 242.600
Epoch: 275, Loss: 273.0117798, Testing Loss: 272.960
Epoch: 300, Loss: 262.1612549, Testing Loss: 249.146
Epoch: 325, Loss: 232.5393982, Testing Loss: 261.607
Epoch: 350, Loss: 229.5829315, Testing Loss: 41.330
Epoch: 375, Loss: 266.6918335, Testing Loss: 241.870
Epoch: 400, Loss: 199.4674072, Testing Loss: 229.663
Epoch: 425, Loss: 212.6084595, Testing Loss: 67.072
Epoch: 450, Loss: 171.1582336, Testing Loss: 225.470
Epoch: 475, Loss: 163.5954590, Testing Loss: 83.846
```

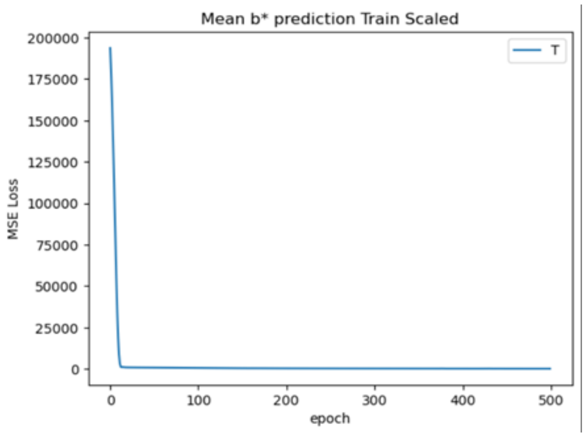
Best MSE achieved on Training set for mean a* prediction: 375.08
Best MSE achieved on Test set for mean a* prediction: 1.04

Best MSE achieved on Training set for mean b* prediction: 163.5
Best MSE achieved on Test set for mean b* prediction: 83.846

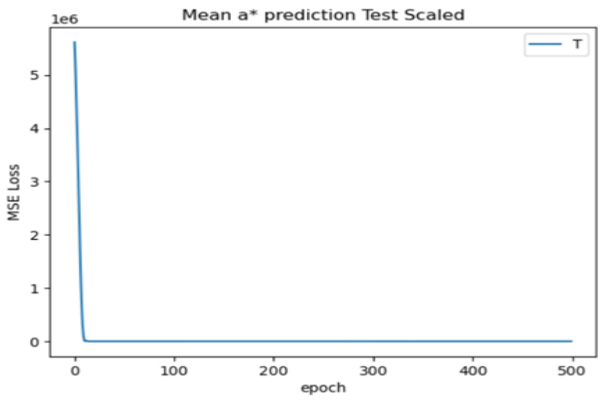
PLOTS:

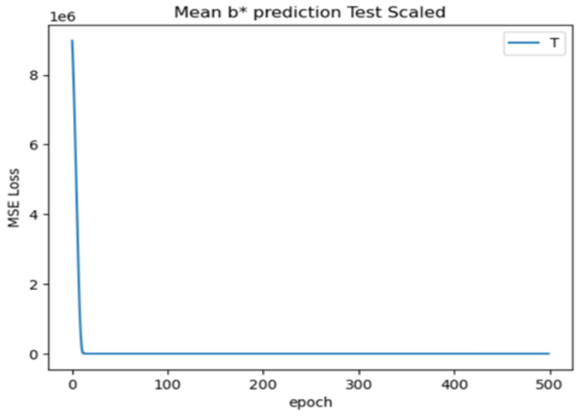
Training MSE:





Testing MSE:





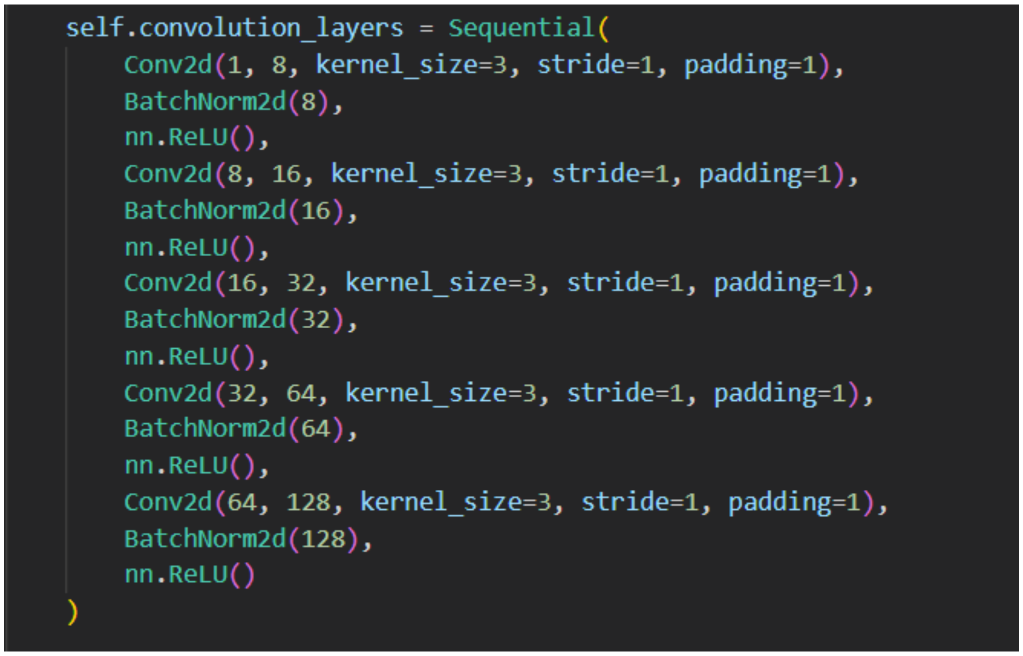
Inference and Conclusion:

CNN outperforms FCN networks for predicting mean chrominance. With CNN, we achieve faster convergence and scaling the input data helps us speed up the training process.

3) Colorization

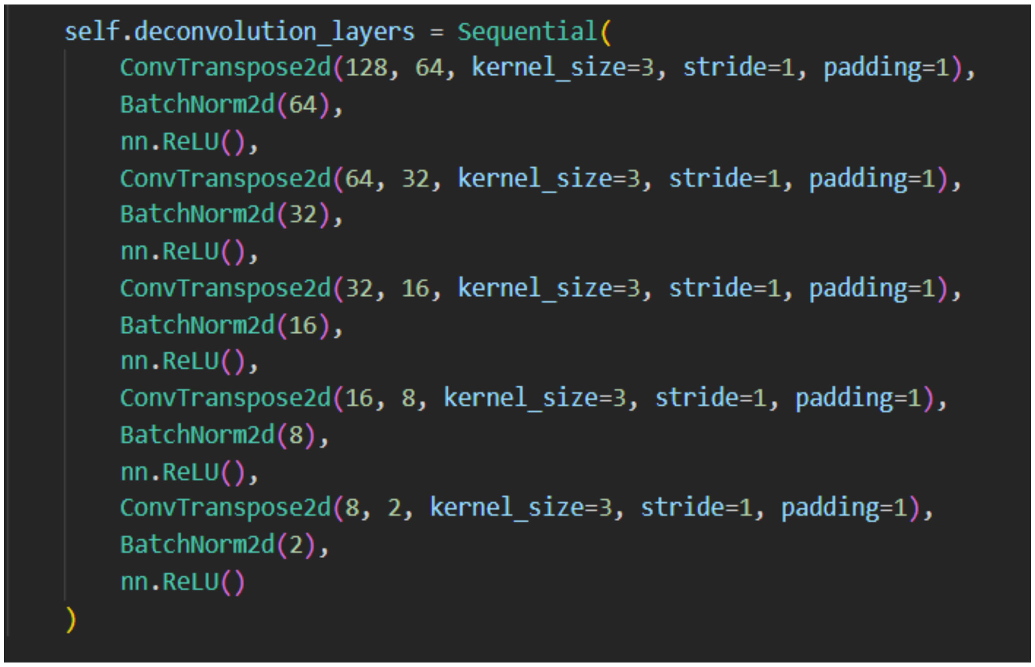
Colorization is the method with which we aim to convert grayscale images to RGB images (colored images). Our input here is of the size 128 x 128 x 1 (taking the L channel). We aim to predict a* and b* channels. We then merge input L* channel, predicted a* channel, and predicted b* channel to generate the colorized images. In order to come up with an optimal architecture that can generalize well, we incorporated downsampling layers and upsampling layers. In our model, we incorporated batch normalization as well. A CNN architecture has been created to pass L channel as input and get a and b channels as output. The error was then propagated backwards and loss is measured in the terms MSE.

Convolution layers:



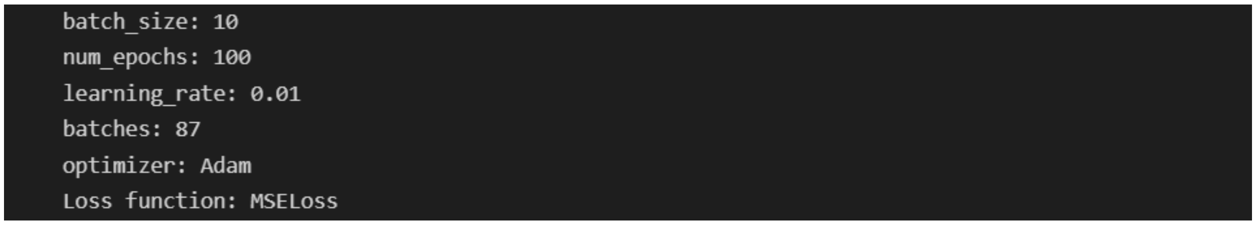
```
self.convolution_layers = Sequential(
    Conv2d(1, 8, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(8),
    nn.ReLU(),
    Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(16),
    nn.ReLU(),
    Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(32),
    nn.ReLU(),
    Conv2d(32, 64, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(64),
    nn.ReLU(),
    Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(128),
    nn.ReLU()
)
```

Deconvolution layers:



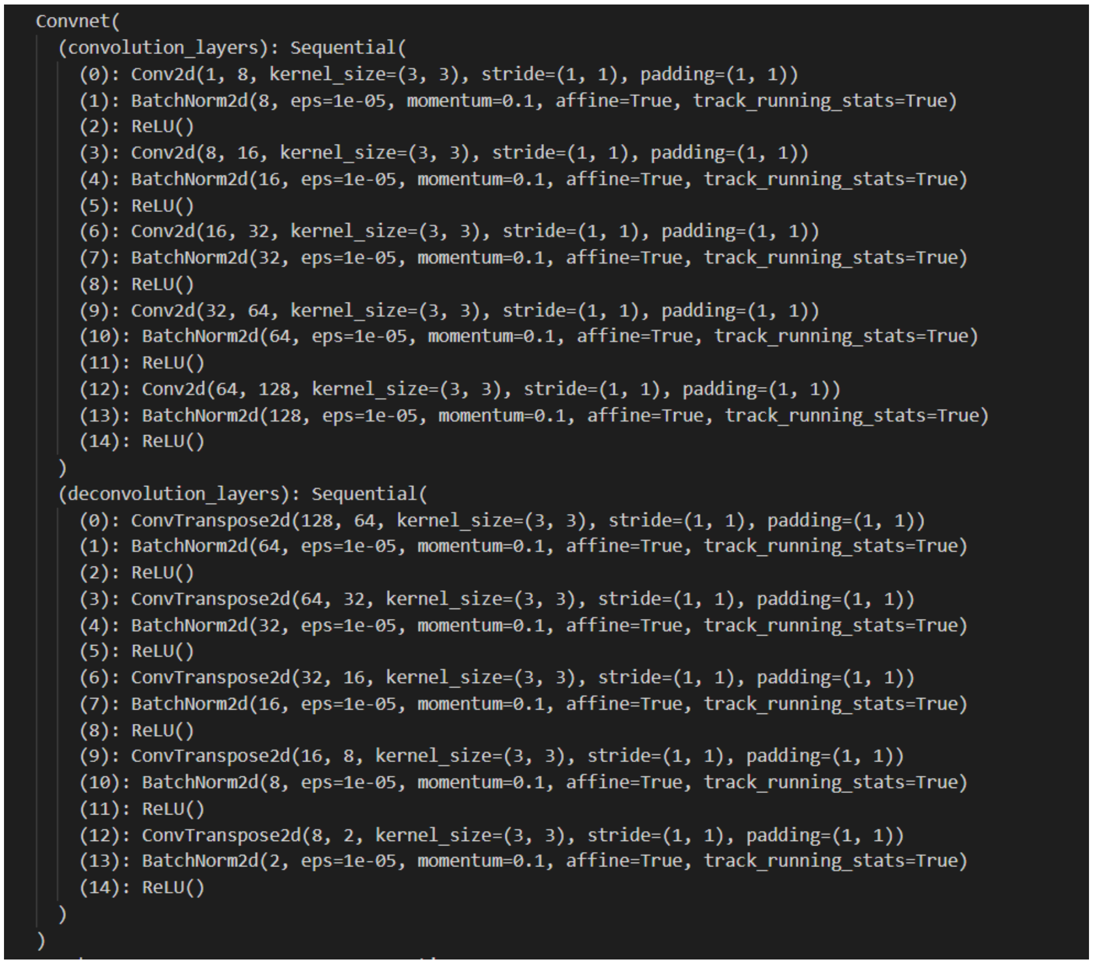
```
self.deconvolution_layers = Sequential(
    ConvTranspose2d(128, 64, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(64),
    nn.ReLU(),
    ConvTranspose2d(64, 32, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(32),
    nn.ReLU(),
    ConvTranspose2d(32, 16, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(16),
    nn.ReLU(),
    ConvTranspose2d(16, 8, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(8),
    nn.ReLU(),
    ConvTranspose2d(8, 2, kernel_size=3, stride=1, padding=1),
    BatchNorm2d(2),
    nn.ReLU()
)
```

Optimal Hyperparamter Setting:



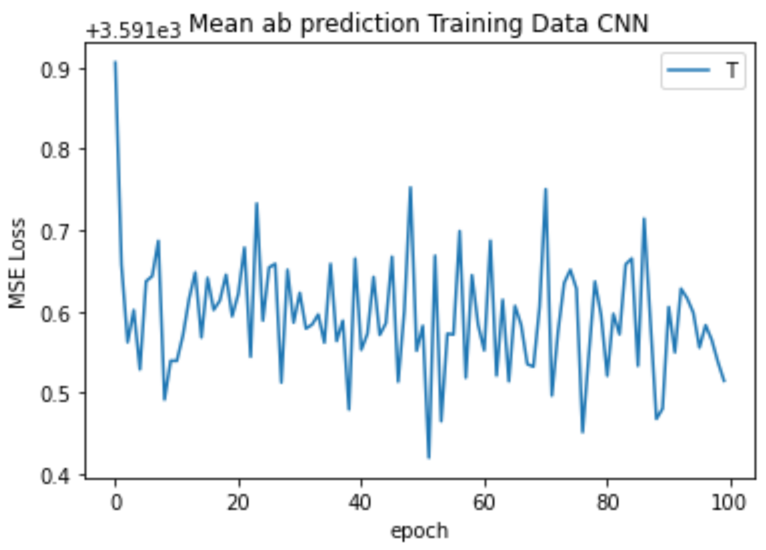
```
batch_size: 10
num_epochs: 100
learning_rate: 0.01
batches: 87
optimizer: Adam
Loss function: MSELoss
```

Best Architecture

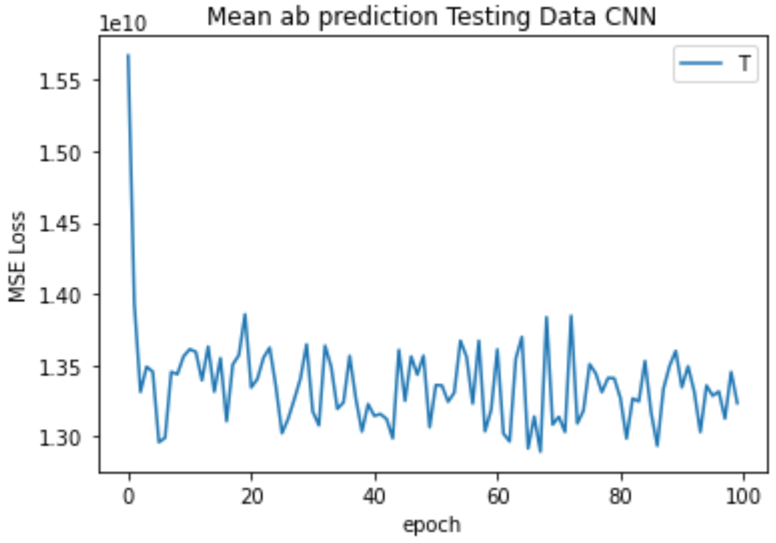


```
Convnet(
  (convolution_layers): Sequential(
    (0): Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): Conv2d(64, 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()
  )
  (deconvolution_layers): Sequential(
    (0): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU()
    (6): ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU()
    (9): ConvTranspose2d(16, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU()
    (12): ConvTranspose2d(8, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU()
  )
)
```

Results after all epochs (training):



Results after all epochs (testing):

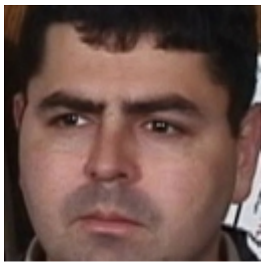


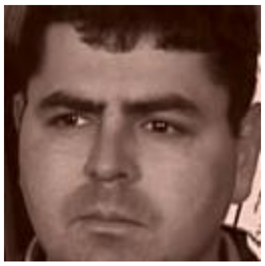
Loss obtained:

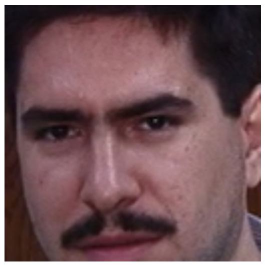
```
Epoch: 0, Loss: 3591.9062500, Testing Loss: 15671934976.000
Epoch: 25, Loss: 3591.6542969, Testing Loss: 13023366144.000
Epoch: 50, Loss: 3591.5825195, Testing Loss: 13360484352.000
Epoch: 75, Loss: 3591.6279297, Testing Loss: 13507350528.000
```

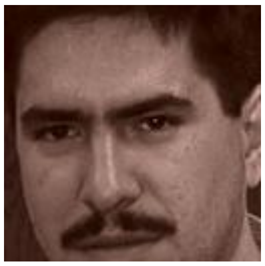
Best MSE achieved on Training set for a*b* channels prediction: 3591.62
Best MSE achieved on Test set for mean a*b* prediction: 13507350528

Colorized Face Images


Original image


Colorized image


Original image


Colorized image

In order to speed up the training process, we used GPU computing. 'colorizer.pkl' file stores the best model weights achieved using this architecture. By scaling the input data, we were able to train on GPUs. Without input data scaling, we were getting out of memory error. We did garbage collection occasionally to clear up the GPUs for training.

More Colorized images can be found in 'colorized_images_train' and 'colorized_images_test' folders.

Evaluation Metrics

Average Peak signal-to-noise ratio (PSNR) value is 28.462568831949948 dB
Average structural similarity index measure (SSIM) value is 0.9243574367606995

Higher PSNR and SSIM indicate that the colorized images are very close to the original images

Improvements

Training on GPUs was not very straightforward. But once we were able to do it, our results improved drastically. Getting the batch_size right turned out to be very important as well. Architectures without downsampling layers didn't give good results.

4) GPU Computing

GPU computing is used to include the GPU (graphics processing unit) as a co-processor to accelerate the training process. The GPU speeds up applications being executed by the CPU by offloading some of the time-consuming portions of the code.

Scaling was done on the input data to avoid out of memory error. The best model achieved by scaling the data is stored in ‘scaled_colorizer.pkl’ file

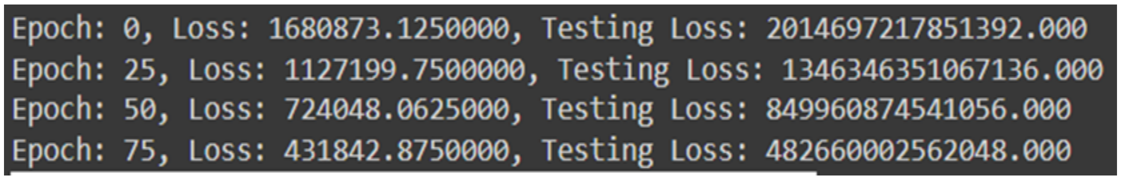
Implementation:

The following commands are executed to use GPU computing:

1. Carry out garbage collection to avoid memory related issues.
   gc.collect()
   torch.cuda.empty_cache()
2. Enable Cuda.
   device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
   Set to cuda:3 for training using the gpu 3.
   device="cuda:3"
3. Verifying CuDNN is enabled
   torch.backends.cudnn.version()
   The above command returns a number that indicates the CuDNN version if it is enabled
4. Finally, the model is saved, the colorized images are generated, and the evaluation metrics are calculated.

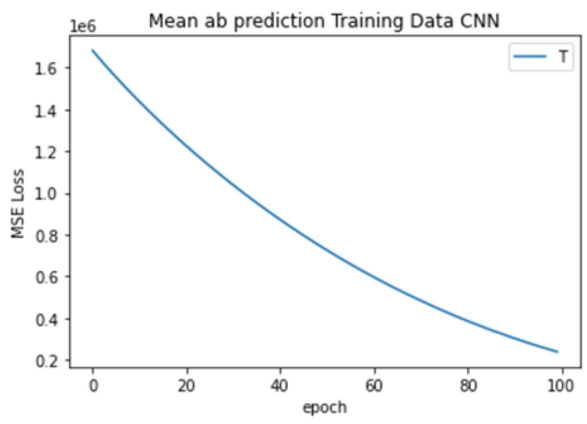
The below observations were made after training the model for 100 epochs.

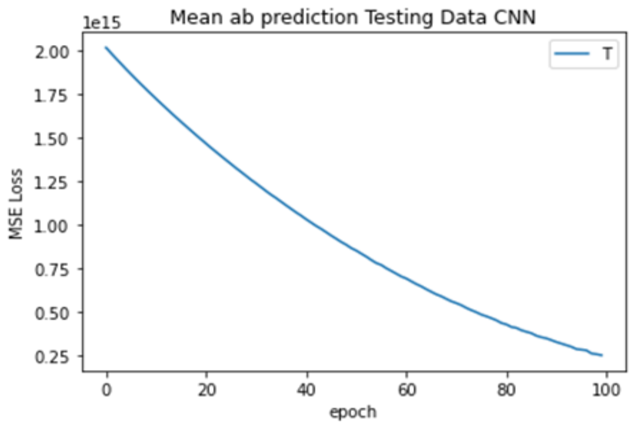
Results:



```
Epoch: 0, Loss: 1680873.1250000, Testing Loss: 2014697217851392.000
Epoch: 25, Loss: 1127199.7500000, Testing Loss: 1346346351067136.000
Epoch: 50, Loss: 724048.0625000, Testing Loss: 849960874541056.000
Epoch: 75, Loss: 431842.8750000, Testing Loss: 482660002562048.000
```

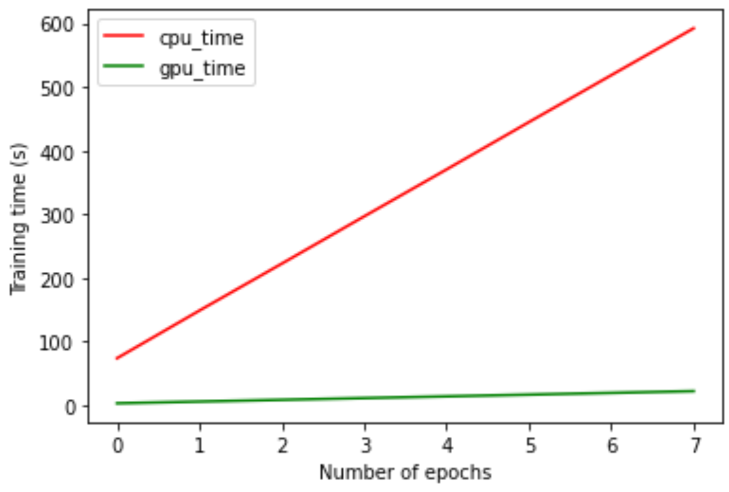
Training and Test MSE loss



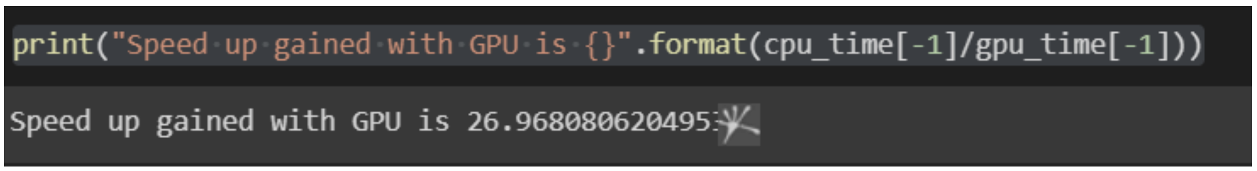


Evaluating GPU Computing:

The same model is trained first using the CPU and then the GPU to compare the total time taken for training. This comparison is then plotted as shown below. The time taken when the GPU is involved is visibly faster than the time taken by the CPU working independently.



Speed up gained with GPU (Extra credit):



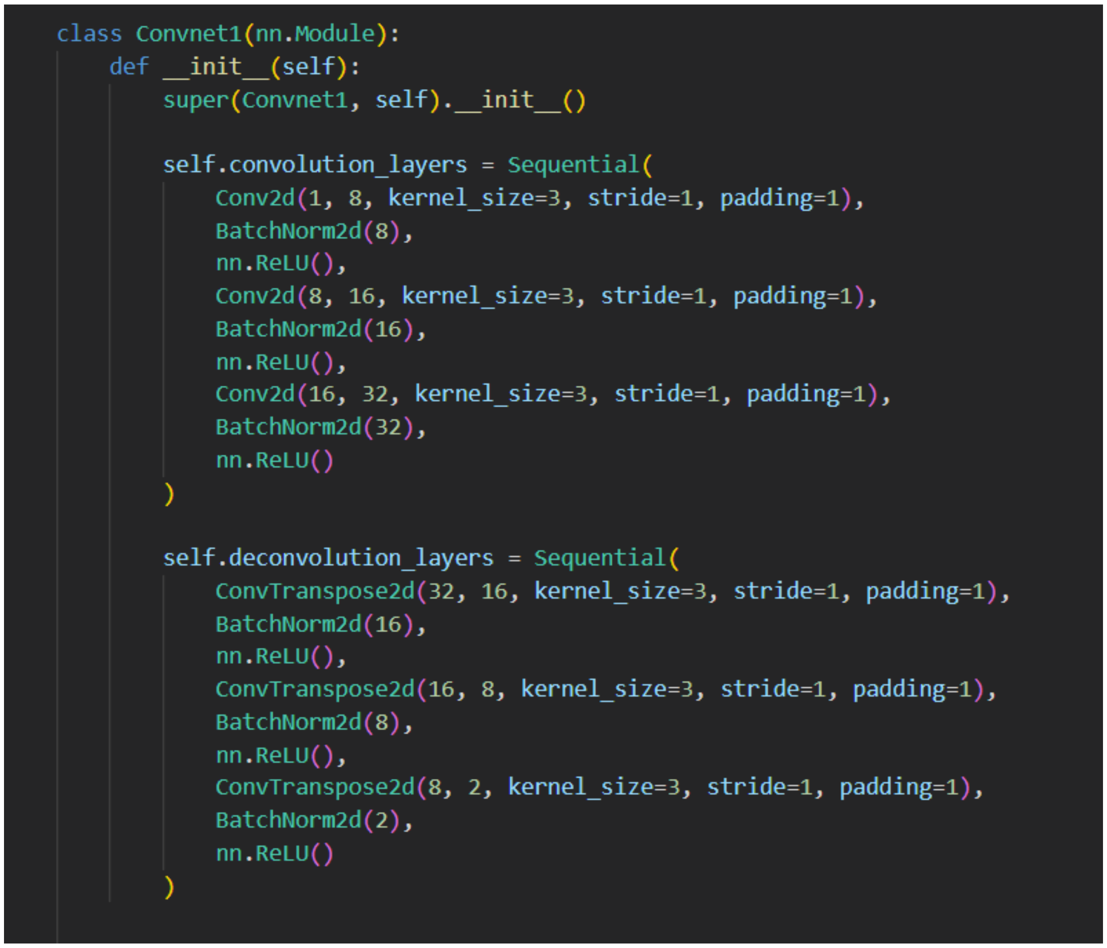
```
print("Speed up gained with GPU is {}".format(cpu_time[-1]/gpu_time[-1]))

Speed up gained with GPU is 26.968080620495
```

As observed, the speed up gained when using the GPU is approximately 27 times that of training using just the CPU.

5) Transfer Learning

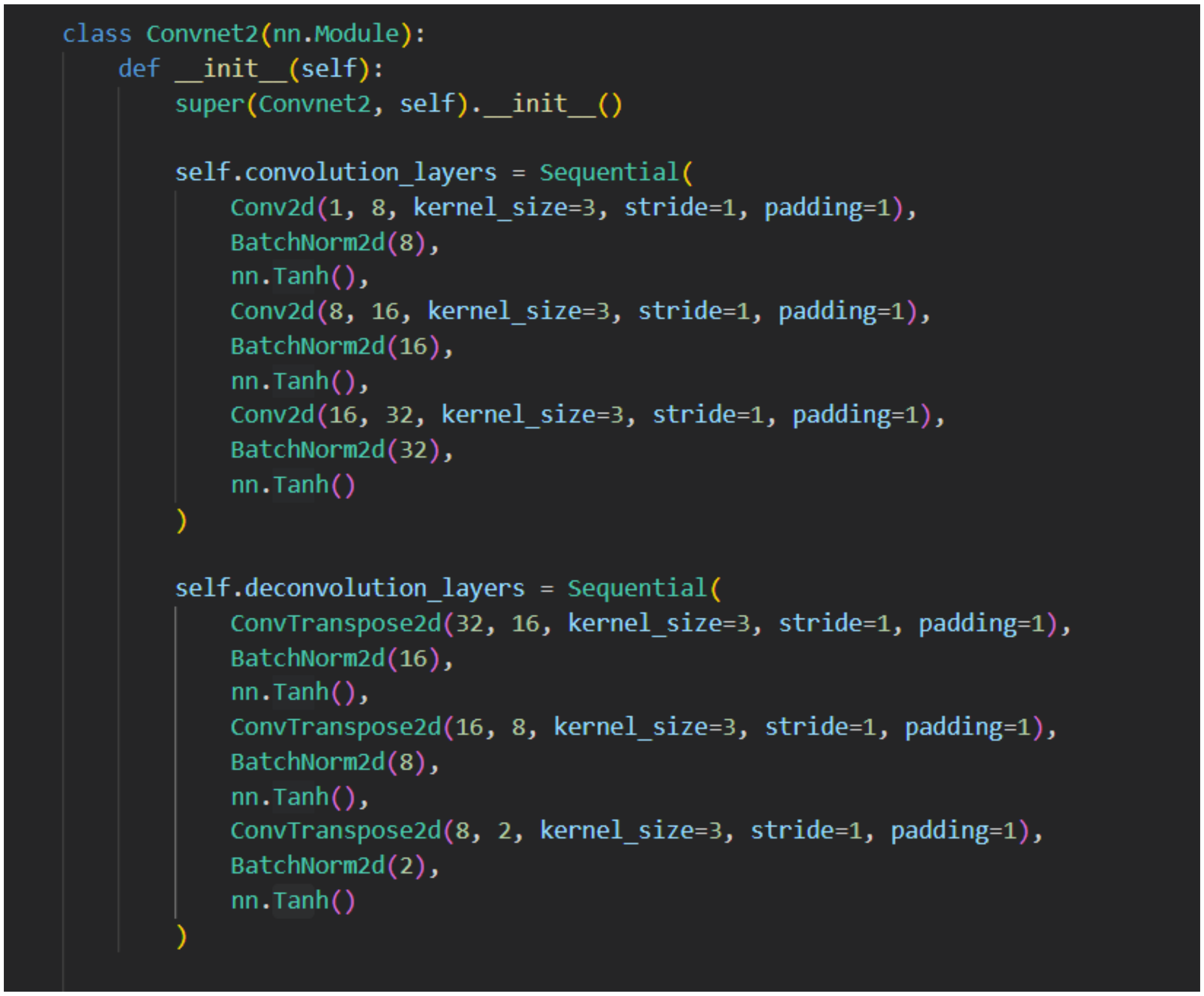
Hyperparameter tuning

Constructing and tweaking model parameters to get the best model architecture is referred to as hyperparameter tuning. Here we have experimented with two architectures to see which one gives the best metrics.



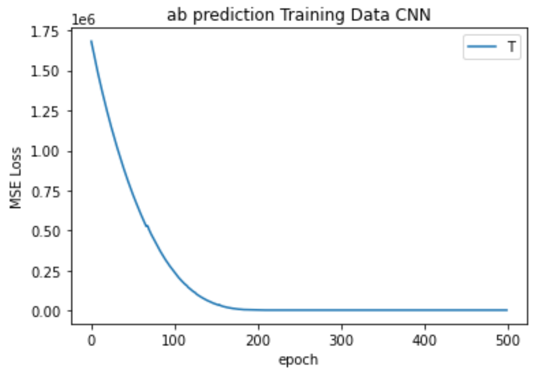
```python
class Convnet1(nn.Module):
    def __init__(self):
        super(Convnet1, self).__init__()

        self.convolution_layers = Sequential(
            Conv2d(1, 8, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(8),
            nn.ReLU(),
            Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(16),
            nn.ReLU(),
            Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(32),
            nn.ReLU()
        )

        self.deconvolution_layers = Sequential(
            ConvTranspose2d(32, 16, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(16),
            nn.ReLU(),
            ConvTranspose2d(16, 8, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(8),
            nn.ReLU(),
            ConvTranspose2d(8, 2, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(2),
            nn.ReLU()
        )
```

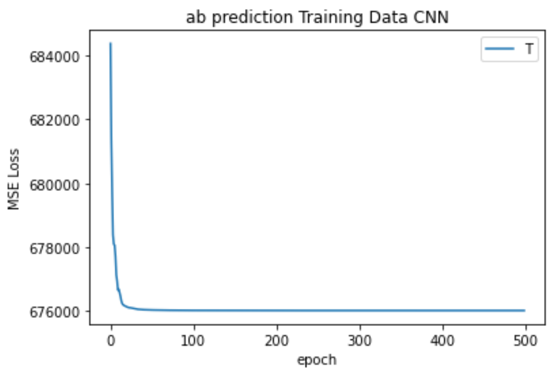
CNN architecture using tanh activation (extra credit):



```
class Convnet2(nn.Module):
    def __init__(self):
        super(Convnet2, self).__init__()

        self.convolution_layers = Sequential(
            Conv2d(1, 8, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(8),
            nn.Tanh(),
            Conv2d(8, 16, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(16),
            nn.Tanh(),
            Conv2d(16, 32, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(32),
            nn.Tanh()
        )

        self.deconvolution_layers = Sequential(
            ConvTranspose2d(32, 16, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(16),
            nn.Tanh(),
            ConvTranspose2d(16, 8, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(8),
            nn.Tanh(),
            ConvTranspose2d(8, 2, kernel_size=3, stride=1, padding=1),
            BatchNorm2d(2),
            nn.Tanh()
        )
```

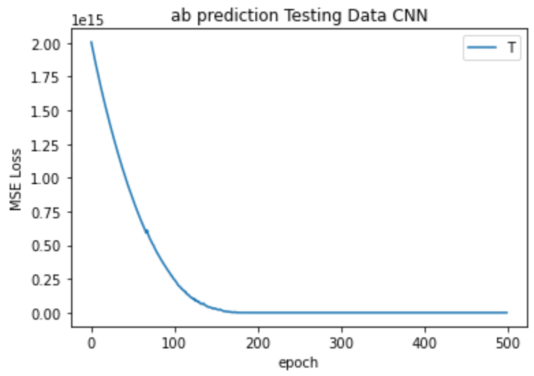
Results:

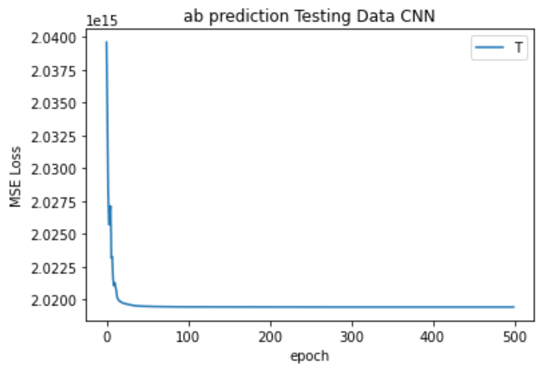




After hyperparameter tuning

After tanh activation





After hyperparameter tuning *After tanh activation*

Transfer Learning

Transfer learning is the concept of using pretrained models for the task at hand by finetuning the pretrained models to genralise on a new task. In this project, the CNN architecture we trained on face images dataset is finetuned for colorizing the images of fruits and vegetables. The advantage of transfer learning is that we need not train all the layers in our network from scratch. We can add new convolution and deconvolution layers for the model to generalise well on NCD dataset and make sure we learn the parameters of only the newly added layers. We set the requires_grad attribute of all the old parameters to False so that we only train the newly added layers.

In this project, we explored two different architectures that makes use of transfer learning. We input the ‘colorizer.pkl’ file we generated in the Colorization part of this project and set require_grad attribute of all the old paramters to False and add new convolution and deconvolution layers.

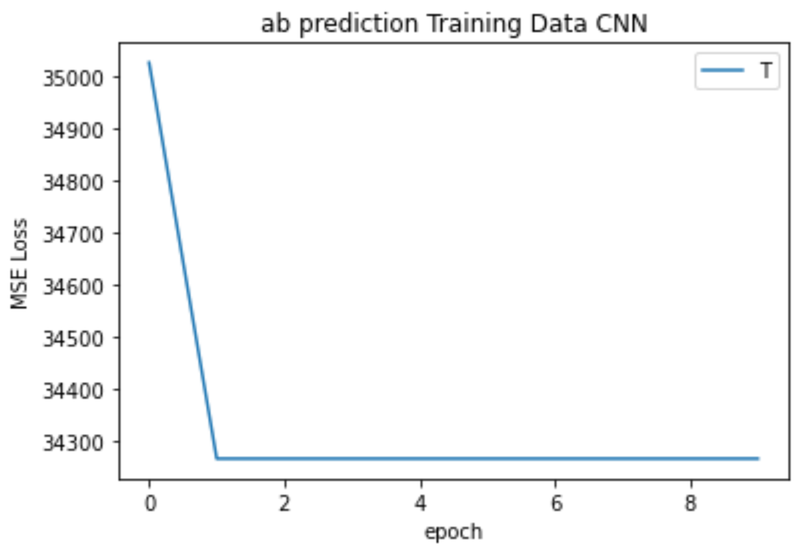
Highlighted cell shows the difference between both architectures

Layer	Architecture 1		Architecture 2	
	Convolution	Deconvolution	Convolution	Deconvolution
(0)	Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	Conv2d(1, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1)	BatchNorm2d(8, eps=1e-05, momentum=0.	BatchNorm2d(64, eps=1e-05, momentum=0.	BatchNorm2d(8, eps=1e-05, momentum=0.	BatchNorm2d(64, eps=1e-05, momentum=0.

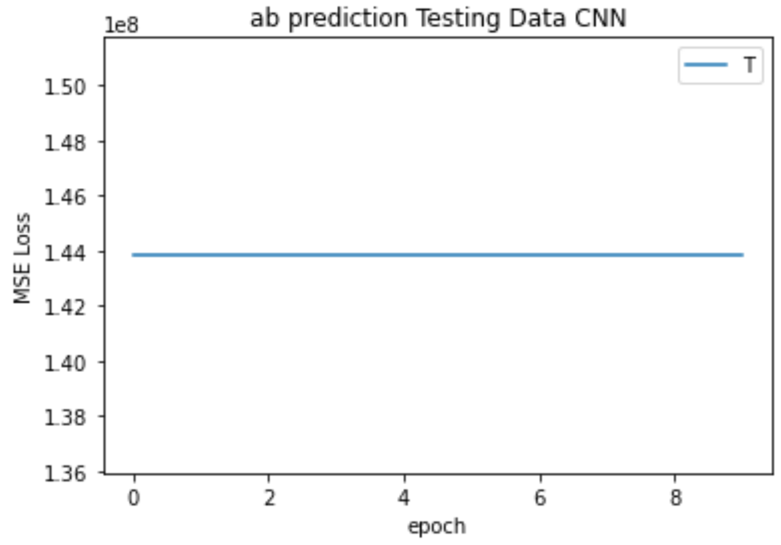
	1, affine=True, track_running_stats=True)	1, affine=True, track_running_stats=True)	1, affine=True, track_running_stats=True)	1, affine=True, track_running_stats=True)
(2)	ReLU()	ReLU()	ReLU()	ReLU()
(3)	Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	Sequential layer with more Conv2d and BatchNorm2d layers nested	Conv2d(8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(64, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(4)	BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(5)	ReLU()	ReLU()	ReLU()	ReLU()
(6)	Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(32, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(7)	BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(8)	ReLU()	ReLU()	ReLU()	ReLU()
(9)	Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(16, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(16, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(10)	BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

(11)	ReLU()	ReLU()	ReLU()	ReLU()
(12)	Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(8, 2, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))	ConvTranspose2d(8, 2, 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)	BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	BatchNorm2d(2, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(14)	ReLU()	ReLU()	ReLU()	ReLU()

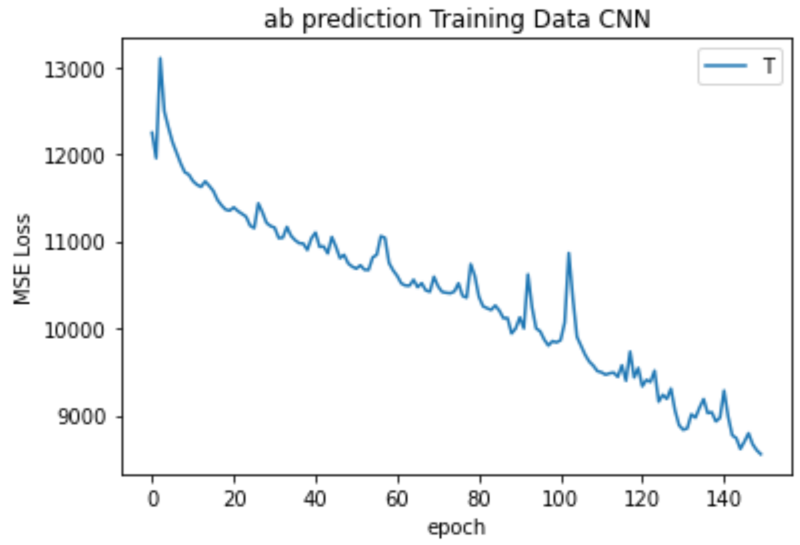
Evaluation of architecture 1 (training data):



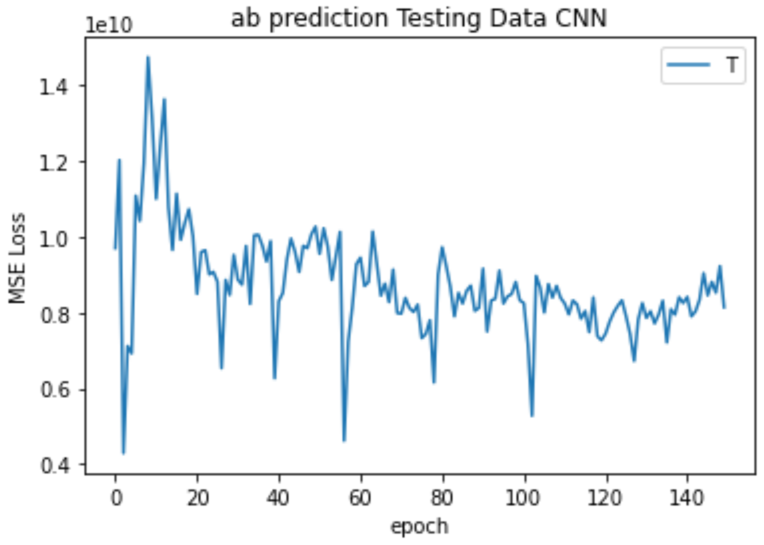
Evaluation of architecture 1 (testing data):

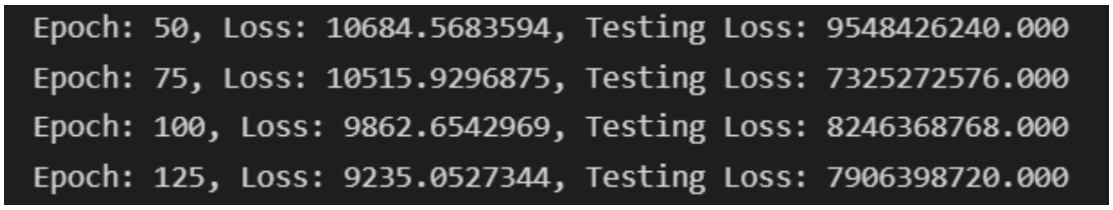


Evaluation of architecture 2 (training data):



Evaluation of architecture 2 (testing data):





```
Epoch: 50, Loss: 10684.5683594, Testing Loss: 9548426240.000
Epoch: 75, Loss: 10515.9296875, Testing Loss: 7325272576.000
Epoch: 100, Loss: 9862.6542969, Testing Loss: 8246368768.000
Epoch: 125, Loss: 9235.0527344, Testing Loss: 7906398720.000
```

Clearly, architecture 2 generalizes better than architecture 1 on the NCD dataset.

Saved output files in transfer learning:
transfer_learning_colorizer_architecture_1.pkl
transfer_learning_colorizer_architecture_2.pkl

Evaluation

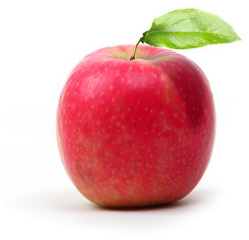
We use PSNR and SSIM metrics to evaluate the performance as underlined below:
PSNR - Peak Signal to Noise Ratio. The higher the PSNR the better the image quality.
SSIM - Structural Similarity Index. SSIM can also be used as a loss function.

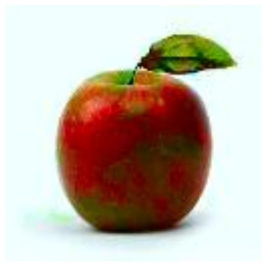
Average Peak signal-to-noise ratio (PSNR) value is 29.997109268611794 dB
Average structural similarity index measure (SSIM) value is 0.7024259505477167

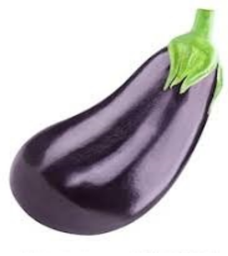
Inference and Conclusion

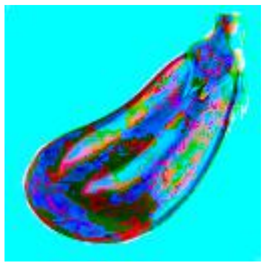
To compare, SSIM obtained on face images is 0.924. While 0.702 correlation is a high value given we trained our original model on face images, it's definitely not state of the art. In future work, we can come up with a better transfer learning approach.

Colorized NCD images


Original image

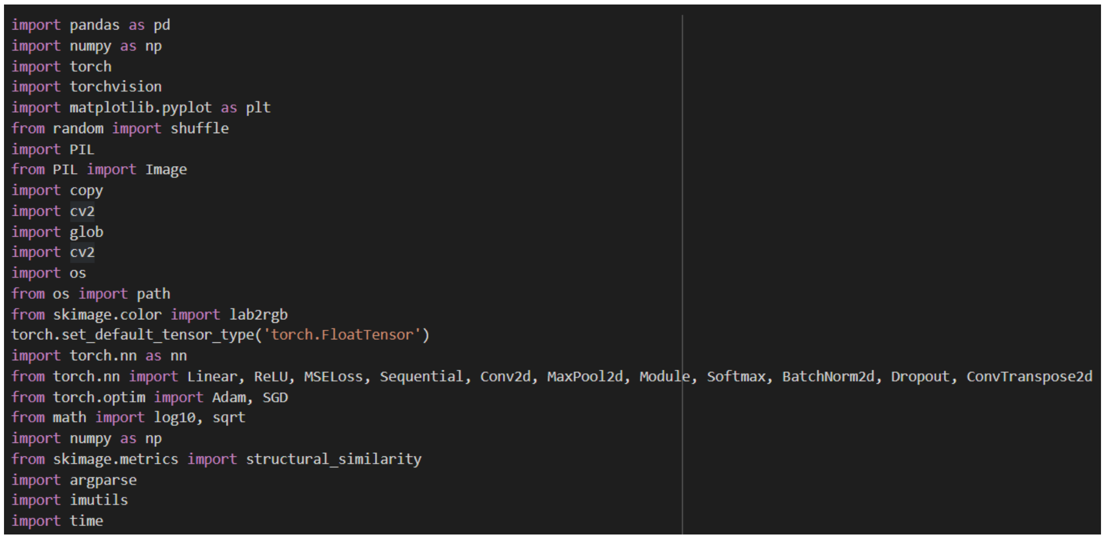

Colorized image


Original image


Colorized image

6) Instructions on how to execute the program

Required Imports and Libraries:


```
import pandas as pd
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
from random import shuffle
import PIL
from PIL import Image
import copy
import cv2
import glob
import cv2
import os
from os import path
from skimage.color import lab2rgb
torch.set_default_tensor_type('torch.FloatTensor')
import torch.nn as nn
from torch.nn import Linear, ReLU, MSELoss, Sequential, Conv2d, MaxPool2d, Module, Softmax, BatchNorm2d, Dropout, ConvTranspose2d
from torch.optim import Adam, SGD
from math import log10, sqrt
import numpy as np
from skimage.metrics import structural_similarity
import argparse
import imutils
import time
```

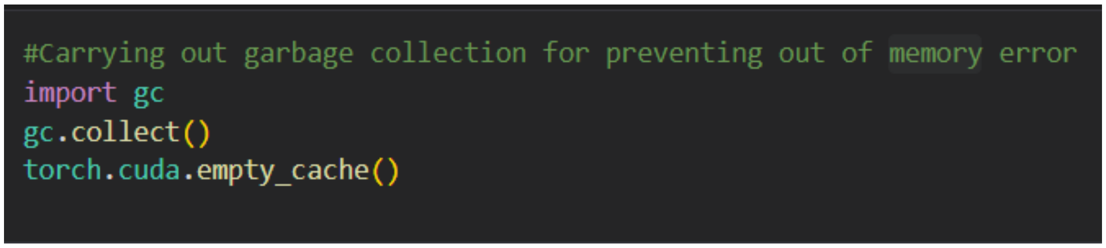
Execution Steps:

1. Import all the necessary files and the datasets. (face_images folder includes Data augmentation images).
2. Import all the trained models (pickle files)

3. Execute the following notebooks as per requirement
    - DL Project 2 Part 1&2.ipynb - Contains data augmentation, Regressor (FCN and CNN) for mean chrominance
    - Colorization (Includes GPU Computing).ipynb - Contains the colorization model and also includes training using GPU computing. Use the last part to colorize images.
    - Evaluating GPU Computing.ipynb - Compare training using the CPU and GPU and calculate the speed up time
    - Hyperparameter tuning.ipynb
    - Transfer Learning.ipynb. Use the last part to colorize images.

7) Challenges

- Out of memory error - The images had to be scaled in GPU computing to avoid the out of memory error. Garbage collection had to be performed to avoid other memory related issues when using GPU computing.



```
#Carrying out garbage collection for preventing out of memory error
import gc
gc.collect()
torch.cuda.empty_cache()
```

- Time - Training times were very high considering the huge number of parameters we try to estimate and optimize in our models. Using GPU computing significantly improved the training times.
- NCD dataset - Applying transfer learning to the model that was trained on the face images dataset did not yield excellent results on the NCD dataset. This might be attributed to the fact that the face images are not as bright as the images of fruits and vegetables in the NCD dataset.

8) References

- https://www.analyticsvidhya.com/blog/2019/10/building-image-classification-models-cnn-pytorch/
- https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce
- https://github.com/torch/nn/blob/master/doc/convolution.md
- http://kbullaughey.github.io/lstm-play/2015/09/21/torch-and-gpu.html
- https://pytorch.org/docs/stable/index.html