Project 2: Deep Image Colorization

Dipali Patidar, Sheela Ippili, Jimmy Ossa

Running Our Code

Unzip our project in a CUDA enabled environment (such as HiPerGator). Run the following files in order:

1. DataAugmentationWithVal.ipynb
2. Regressor_training_testing.ipynb
3. ColorizeEarlyStop.ipynb

Installing our Custom Kernel

Note: Step three may require that you use the same conda environment as us. Included in our deliverables is a **req.txt** that you can use to generate a conda environment identical to ours. Our **req.txt** file is specific to Linux.
Note: If using HiPerGator, you will need to run “module load conda” in HiPerGator so that conda will be available.

Use the following command to create a conda environment from our **req.txt** file:

```
conda create --name colorize_net_custom --file req.txt
```

Use the following command to convert the conda environment to a Juptyer Notebook kernel:

```
python -m ipykernel install --user --name=colorize_net_custom
```

You should now be able to see our kernel in Jupyter notebook and use it to run step three (this works in HiPerGator)

Note: Steps one and two can be run using the default **PyTorch-1.8.1** kernel in HiPerGator.

Loading the images, Training and Data Augmentation

We first loaded the images using open cv, os and glob from the dataset folder which has 750 images. We shuffled the images randomly and divided the images into train test and test set in the ratio of 9:1 where the train set has 675 images and the test set has 75 images. Then the train set is augmented by applying different types of transforms such as horizontal and vertical flips, random crops, and random rotation using PyTorch compose and transform functions. After augmentation, there are 6750 images in the train set and 75 images in the test set. The images

are then turned into tensors and are also normalized for converting them from RGB to LAB space, for applying the regressor, and colorizing the images.

Regressor

The Model

The regressor model takes grayscale **L*** channel as input generated from augmented data and predicts the mean chrominance values i.e channels **a*** and **b*.** This model is also responsible for downsampling the image. The model consists of 12 layers in total, each 2d- convolution layer followed by batch normalisation layer and non-linear leaky relu function. The configuration of the regressor and colorizer are specified in the table below. The number of channels of the feature maps are motivated by the Resnet 18 architecture. The entire model is built using pytorch. The hyperparameters, using which we have trained the model after experimenting are listed down in the second table.

Layer	Type	Depth	Kernel	Stride	Padding
1	Convolution	32	4*4	2	1
2	BatchNorm	32	-	-	-
3	Convolution	64	4*4	2	1
4	BatchNorm	64	-	-	-
5	Convolution	128	4*4	2	1
6	BatchNorm	128	-	-	-
7	Convolution	256	4*4	2	1
8	BatchNorm	256	-	-	-
9	Convolution	256	4*4	2	1
10	BatchNorm	256	-	-	-
11	Convolution	512	4*4	2	1
12	BatchNorm	512	-	-	-

Fig 1: Regressor model architecture

Regressor HyperParameters

Parameter name	Value
Learning rate	0.0001
Epoch	200
Weight decay	1e-5

Fig 2: Regressor model hyperparameters

The Model PyTorch Code

```
  feature_maps = nn.Sequential(
           # input is Z, going into a convolution
           nn.Conv2d(in_channel,32,4,2,1),
           nn.BatchNorm2d(32),
           nn.LeakyReLU(),

           nn.Conv2d(32,64,4,2,1),
           nn.BatchNorm2d(64),
           nn.LeakyReLU(),

           nn.Conv2d(64,128,4,2,1),
           nn.BatchNorm2d(128),
           nn.LeakyReLU(),

           nn.Conv2d(128,256,4,2,1),
           nn.BatchNorm2d(256),
           nn.LeakyReLU(),

           nn.Conv2d(256,256,4,2,1),
           nn.BatchNorm2d(256),
           nn.LeakyReLU(),

           nn.Conv2d(256,512,4,2,1),
           nn.BatchNorm2d(512),
           nn.LeakyReLU(),
       )

     y_hat = torch.sigmoid(self.lin(feature_maps.reshape(-1, 512 *2* 2)))
```

Fig. 3 Regressor Network PyTorch Code Definition

Loss Function Plot

The regressor training MSE received is

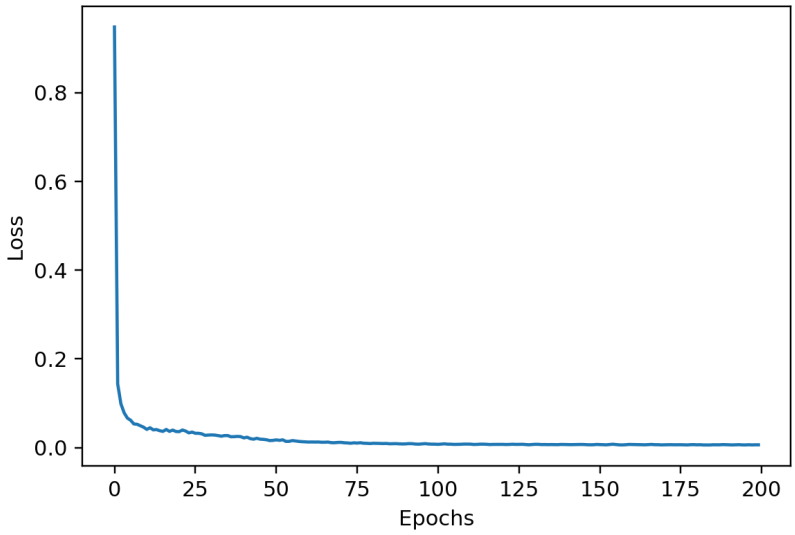


Fig. 4 MSE loss vs epoch curve for Regressor

Regressor Output

The regressor outputs two scalar values which are the average of a and b for each input image. The average **Test data MSE** value is **3.086001379415393e-05.** Value of mean chromiance for 4 test images are tabulated below.

Image_num	Mean a	Mean b
Image: 1	12.4583	9.7027
Image: 2	11.9247	12.0544
Image: 3	14.2982	3.2070
Image: 4	10.2683	7.8804

Fig. 6 Mean a and Mean b value predicted by model on test data

Colorization

The model

The model takes as input the grayscale **L** channel from the input image and outputs two channels, the predicted **A** and **B** channels. Our loss criterion is the mean squared error between the ground truth LAB colorspace image, and the predicted LAB colorspace image (by merging the input **L** channel with the output **A** and **B** channels). To manually inspect the images, we convert the network outputs to the RGB colorspace and compare with the original ground truth images.

Our colorization network consists of six layers divided into two groups, a downsampling group and an upsampling group. The downsampling layers **(fig. 7)** consist of three 2-dimensional convolutional layers, and the upsampling layers **(fig. 8)** consist of three 2-dimensional transpose convolutional layers. Each convolutional layer is followed by a 2-dimensional batch normalization function, and a non-linear ReLU activation function. Below is a table describing the convolutional layers of our network.

Downsampling Layers				
Type	Kernel	Stride	Padding	Output Channels
Conv. 2d	3x3	2	1	8
Conv. 2d	3x3	2	1	32
Conv. 2d	3x3	2	1	128

Fig. 7 Colorize Net Downsampling Convolutional Layers

Upsampling Layers				
Type	Kernel	Stride	Padding	Output Channels
Conv. Transpose 2d	2x2	2	0	32
Conv. Transpose 2d	2x2	2	0	8
Conv. Transpose 2d	2x2	2	0	2

Fig. 8 Colorize Net Upsampling Convolutional Layers

We tested with different numbers of upsampling/downsampling layers (N = 4, N = 5, N = 6) and found that 3 layers in both the downsampling and upsampling groups produced results that we considered to be the most visually accurate as well as a low MSE loss value. We choose a learning rate of 0.01 and a decay of 0.0 **(fig. 9)**. Our model trains for a maximum of 20 epochs, however it will stop early if the model doesn't demonstrate improved performance on the

validation for five consecutive epochs. Note: to implement early stopping, we used a python library called PyTorch-Ignite, which is PyTorch recommended third party library for early stopping.

Learning Rate	Decay	Max Epochs
0.01	0.0	20

Fig. 9 Colorize Net Training Hyperparameters

The Model PyTorch Code

Below is a PyTorch code snippet which defines our network layers.

```
self.downsample = nn.Sequential(
    nn.Conv2d(1, 8, 3, 2, 1),
    nn.BatchNorm2d(8),
    nn.ReLU(),
    nn.Conv2d(8, 32, 3, 2, 1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Conv2d(32, 128, 3, 2, 1),
    nn.BatchNorm2d(128),
    nn.ReLU()
)

self.upsample = nn.Sequential(
    nn.ConvTranspose2d(128, 32, 2, 2),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.ConvTranspose2d(32, 8, 2, 2),
    nn.BatchNorm2d(8),
    nn.ReLU(),
    nn.ConvTranspose2d(8, 2, 2, 2),
    nn.ReLU()
)
```

Fig. 10 Colorize Net Network PyTorch Code Definition

GPU, Dataloader, and Early Stopping Optimizations

We ran our code as a Jupyter Notebook on the HiPerGator server with an A100 GPU. We moved all tensors to the GPU during training to accelerate training speed. Additionally, we created a custom dataset loader so that our code only loads in enough images for a single mini-batch. We found that without a custom dataloader, loading all augmented images simultaneously required too much memory and would crash our kernel. The custom data loader improved the memory efficiency of our program dramatically.

We also implemented an early stopping mechanism to prevent overfitting. PyTorch does not natively support early stopping, but the PyTorch recommended approach is to use an external package called “Ignite”. We configured Ignite to run at most 20 epochs, and to stop early if the model doesn’t make any mean squared error loss progress after five epochs. Ignite calculates the early stopping loss on the validation set between each epoch. To support this mechanism, we divided our test set in half: into a test and validation set.

Results

With early stopping enabled, our network would always stop between five and ten epochs **(fig 11 and 12)**. Below are graphs showing the training loss performance of our network using a "ReLU" final layer activation function and "Sigmoid" final layer activation function, respectively.

ReLU Final Layer Activation Function:

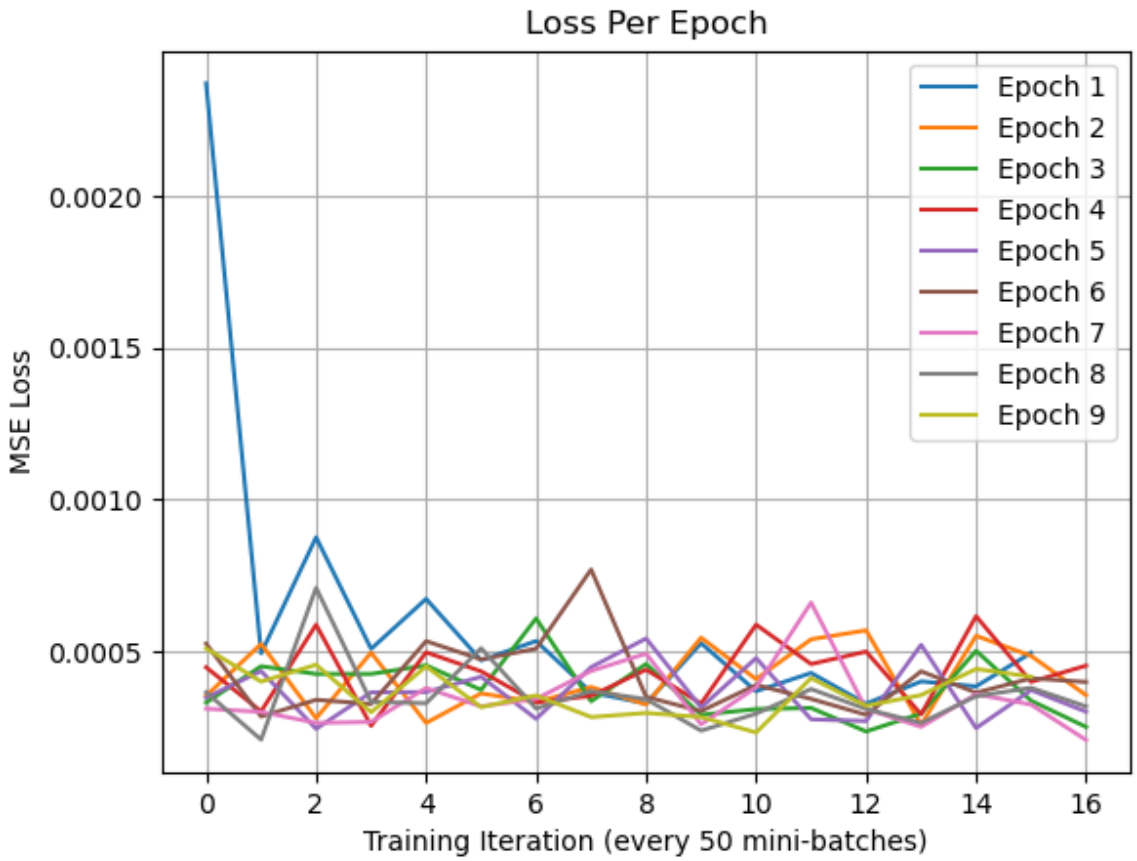


Fig. 11 Epoch performance with ReLU output layer

Sigmoid Final Layer Activation Function:

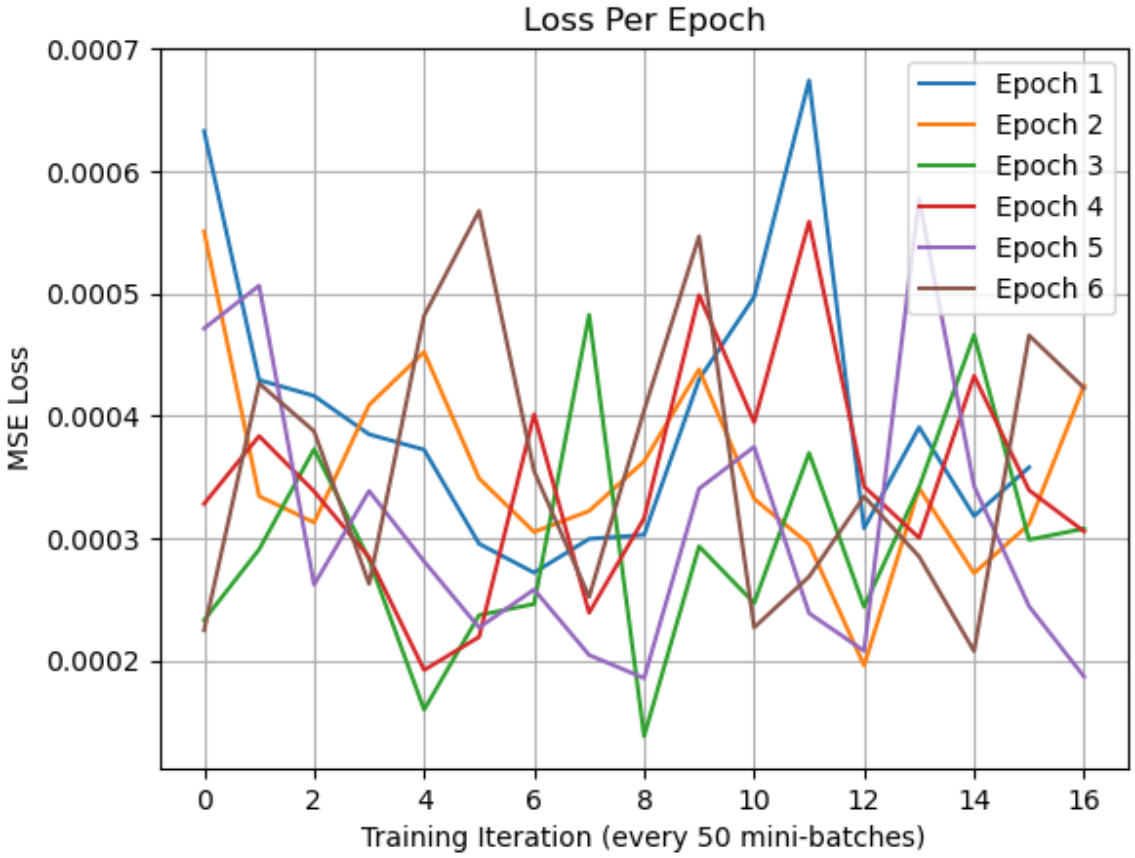


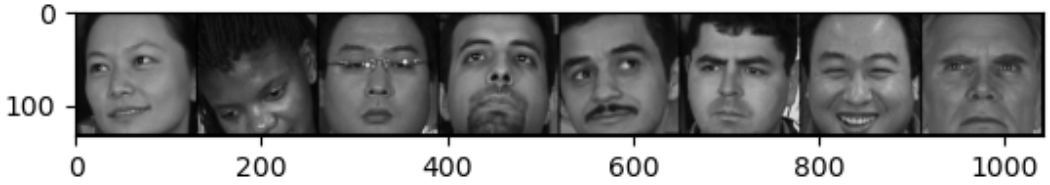
Fig. 12 Epoch performance with Sigmoid output layer

We found that the network would show only minimal improvements with each subsequent epoch until eventually stopping early. The “ReLU” final layer activation function **(fig. 11)** had a higher MSE loss in the first epoch and converged after nine epochs, whereas the “Sigmoid” activation function **(fig. 12)** had a lower initial loss and converged after six epochs, three less epochs when compared to “ReLU”.

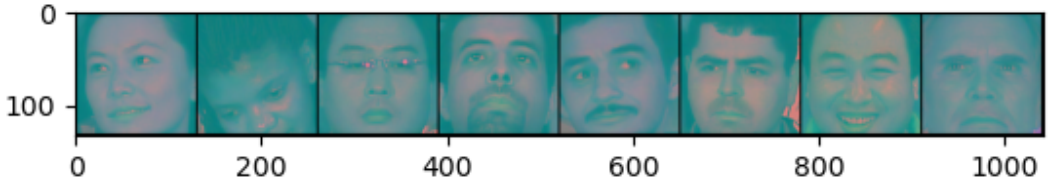
Colorized Images

Below are examples from a single test batch of our network's performance using the Sigmoid activation function in the output layer.

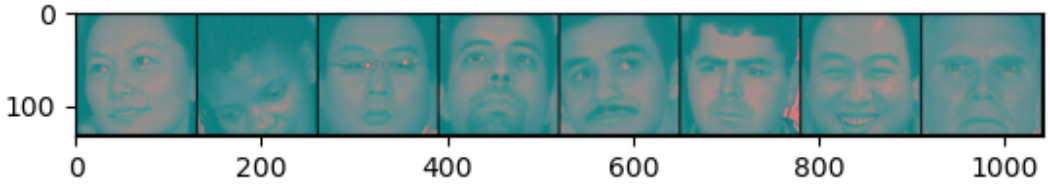
Input Grayscale:



Ground Truth LAB Colorspace:



Output LAB Colorspace:



Ground Truth RGB Colorspace:

Output RGB Colorspace:

Fig. 13 Test min-batch ground truth compared to outputs

Mean Squared Loss for Above Samples:

Sample 1	Sample 2	Sample 3	Sample 4	Sample 5	Sample 6	Sample 7	Sample 8
0.00074	0.00024	0.00017	0.00028	0.00043	0.00017	0.00043	0.00033

Fig. 14 MSE loss for each test sample in **fig. 13**

Average Mean Squared Loss over Entire Test Test:
0.00038

Alternative Output Layer Activation Functions

We tested the performance of our model using three different output layer activation functions. Below are the average MSE loss results of these activation functions. Note that the input tensors were normalized in the range of 0 to 1 for all tests.

	ReLU	**Sigmoid**	**Tanh**
Average MSE Loss	41e-5	38e-5	46e-5

Fig. 15 Average MSE loss for each type of output activation function

Discussion

The “Sigmoid” final output activation function produces the lowest average MSE loss when given input tensors normalized in the range of 0 to 1 **(fig. 15)**. We observe that the colorized outputs of our network generally apply color to the faces in the pictures, while leaving the background unchanged. However, the colors applied to the faces don’t represent the full diversity of the input dataset. The output images appear dull when compared to the ground truth, and we notice that common features such as lip color aren’t being captured by our model.

A possible explanation for this is that our MSE loss criterion should be modified to place a greater emphasis on facial features. For example, if our loss criterion prioritizes color differences around facial landmarks over, say, color difference in the background, then the model may better identify the nuanced differences in how facial features correspond to color (such as the color red of someone’s lips). Additionally, it is worth mentioning that this model is very simple, so it is unlikely that our model will be able to capture the complexity and variety of how facial structures correspond to color. It is possible that our model is simply making a “best guess” by averaging the colors of all input images, and this may explain how the output images are more similar in color than the input images.

Optional Credit

Is classifier better than regressor for colorization of gray scale images?

The paper states that a classification task is used to colorize a grayscale image. It can be observed that a classification task works better than a regressor. By using a classifier, we can classify a particular part of the image and color the image based on the label but by using a regressor we have to colorize the image as a whole. For example, we find a pattern and if it contains grass, we know that grass is generally green in color and this leads to classification task where we know that if grass is identified in the picture, we colorize that part of the region green thus making things simpler. But with a regressor, we have to colorize the whole image based on the patterns. Therefore, a classifier works better than a regressor for image colorization.

Explore changing the number of feature maps for the interior CNNs to see if you can gain better test accuracy.

For Colorization of images, we increased the max number of channels in the downsampling layer from 128 to 256 and observed an MSE loss of 36e-5, a marginal improvement over the original MSE loss of 38e-5. We further increased the max channels to 512 and observed an MSE loss of 35e-5, again only a marginal improvement. Next, we increased the number of downsampling and upsampling convolutional layers to five layers in each group (10 total) and yielded an MSE loss of 40e-5 which suggests that increasing the model size and features beyond this point will no longer improve performance.

We also evaluate the model with a fewer number of features. We change the maximum number of channels to 64 (still using 10 total convolutional layers) and observe an MSE loss of 39e-5. The result of these different configurations is that the model with 6 total convolutional layers, and a max number of channels equal to 512, performed the best on our test dataset.