Deep Learning Project Part2 Report

Nayanika Ghosh 4191 4976| Disha Nayar 6199 1035

In this project, we accomplish automatic colorization for black and white photos using a convolutional neural network. Below we describe in detail the steps to run and different stages of our project.

Steps to Run on Hypergator : Our submission consist of a zip folder with 3 Jupyter notebooks, 1 Dataset *face_images* folder and Pre-trained model files (*Network3.pt* and *Tanh.pt*) that can be directly used for testing.

1. Unzip Project2.zip
2. Run Simple Regressor:
    a. Open *Regressor_train.ipynb* – This notebook has three sections. Data Segregation, Data loading and Augmentation, Network definition and Training.
    b. To run the file, click Run -> Run all cells.
    c. To view the results, scroll to the training section and see the loss while training for each epoch.
    d. After the model is trained, you can view the graph depicting the training loss vs epochs.
3. Run Colorization:
   Training –
    a. To train the Colorization network, open *Colorization_train.ipynb* – Contains the same sections as in simple regressor.
    b. To run the file, click Run -> Run all cells.
    c. To view the results, scroll to the training section and see the images generated at each epoch.
    d. After the model has trained, you can view the graph depicting the training loss vs epochs.
   Testing –
    a. For testing the Colorization network, open *Colorization_test.ipynb*. This file loads the Network3.pt pre trained model.
    b. To run the file, click Run -> Run all cells.
    c. Scroll to the testing section to view the results.
4. Optional Credit 1 (using Tanh):
   Training –
    a. Change useTanh =True in *Colorization_train.ipynb*
    b. Rest of the steps are same as training in point 3.
   Testing –
    d. Change useTanh = True in *Colorization_test.ipynb*. When useTanh is set to True, this file loads the pre trained model Tanh.pt for testing.
    e. Rest of the steps are same as testing in point 3.

Loading Dataset : For this project, we use *face_images* as our base dataset. Next, we shuffle the images and split them into train and test images in 90:10 ratio. With our dataset, we get 675 train and 75 test images. These images are stored in the Train & Test folders respectively.

We are using pytorch function DataLoader to load the train and test dataset. The DataLoader takes in a custom defined class *CustomDataset* for resizing the images to 128x128. This custom class also generates 10 augmented images per training image as detailed out in the Augment data section. The data is shuffled by setting shuffle=True in DataLoader.

Augment Data : To increase the amount of training data, we augment our train set images.

1. We are using pytorch's transforms.Compose function to augment data. Each image in the training set undergoes random horizontal flip, random crop and random scaling between [0.6, 1.0].
2. For performing the transformations defined above, we have written a custom class ColorDataset which takes in the input dataset, along with the transforms and generates the 10 augmented data per input train images, totalling to 6750 images.

3. For Simple regressor the *ColorDataset* class returns L and mean values of a and b for each image. For Image colorization, it returns L, a and b channels for each image. The L channel is normalized to lie between [0,1].
4. Finally, the training data is then loaded in mini batches of 100 using a torch DataLoader. The images are then resized to 128x128 before it is passed to the network.

Convert Images to LAB : The class *ColorDataset* is also responsible for converting all the RGB images into LAB space. We are using Opencv *cvtColor* function to convert the images to L, a and b channels.

Simple Regressor : We create a simple regressor to predict the mean chrominance values for the input images.

1. We first define a RegNetwork class of nn.module type to define our network. We have 7 Conv2d layers each of which is followed by a LeakyReLU activation function, except the last layer.
2. The number of feature maps remain 3 at all the hidden layers except at the output layer, where we output 2 channels. Further we are using a kernel size of 3. We use a stride of 2 and padding as 1 to decrease the size of our output by half at each layer.
3. The loss for our model is calculated using a MSEloss and we are using Adam optimizer to train the model, with a learning rate of 0.001.
4. After we have our model ready, we train our model for 200 epochs and notice that the training error reduces with each iteration.

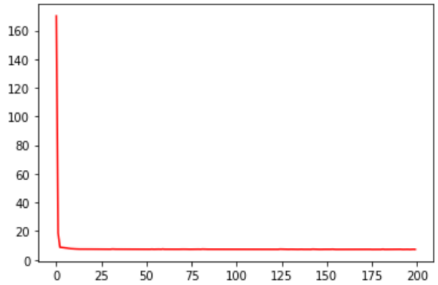


Figure 1 : Graph for training loss for 200 epochs. Y axis represents the loss. X axis represents the number of epochs.

Colorize the image : We have defined and trained a network to predict a and b chrominance channels to get a coloured image.

1. We first define a *ColorNetwork* class of nn.Module type to define our network. Our network is divided into downsampling and upsampling modules.
2. We have 5 downsampling modules, each of which has a conv2d layer, followed by batch normalization and then a LeakyReLU activation function. In each module we are reducing the size of the image by 2. Further, as we are going deeper into the network, we want to learn more complex features and hence we increase the number of feature maps in the downsampling modules as 16,32,64,128, 256.
3. These layers are followed by upsampling modules. Each of these modules have convtranspose2d for performing deconvolution, which is followed by batch normalization and LeakyReLU layers. In upsampling, we are reducing the feature maps as 256,128,64,32,16,2.
4. In each layer of the network we are using a stride of 2 to reduce the output size by half in downsampling and increase the size by double in upsampling module. Kernel size is taken as 3. The loss of this model is calculated using MSEloss and we are using Adam optimizer with a learning rate of 0.01 to train the model.

Training : After data loading & augmentation. Our training dataset consists of 6750 images. We train our model on these images and output the loss in each epoch followed by the predicted output, a and b channels combined with the input L. The model is trained for 200 epochs. Figure 2 shows the loss at different epochs during training. It also displays the grayscale image, followed by the output predicted by our model and the ground truth RGB image. Figure 3 shows the graph for training loss.

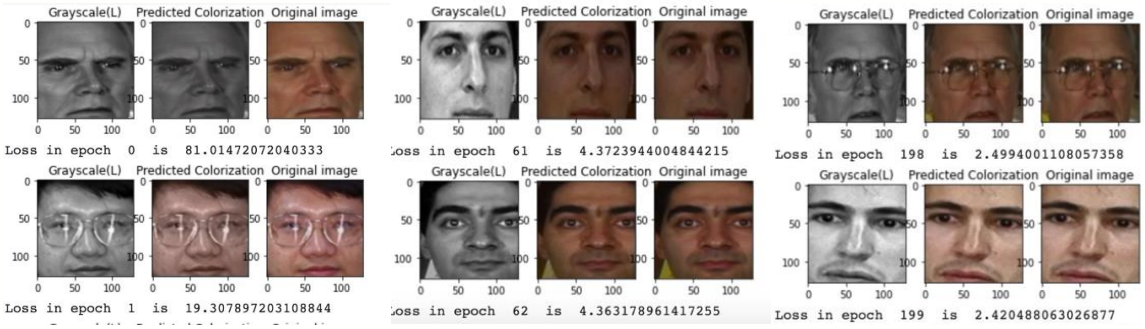


Figure 2 : Left image shows the result from epochs 0 and 1. Middle image shows intermediate training output, i.e the results from epochs 61 and 62. The right image shows results for epochs 198 and 199. The training loss is decreased with each epoch as the model learns.

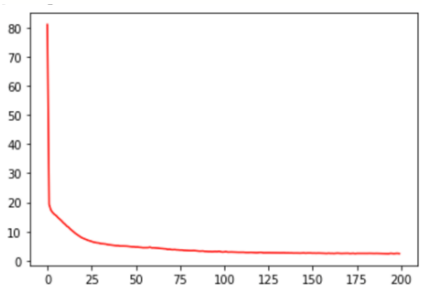


Figure 3: Graph for training loss for 200 epochs. Y axis represents the loss. X axis represents the number of epochs.

Testing : 10% of the original dataset is used for testing our model. To prevent allocating too much memory to hidden layers, we have performed our testing in mini batches of 10. Further, we are also using model.eval() before predicting the output of model for test images. This tells the batch-normalization layers that the model is in evaluation stage. The average loss for all the test images is 22.19.

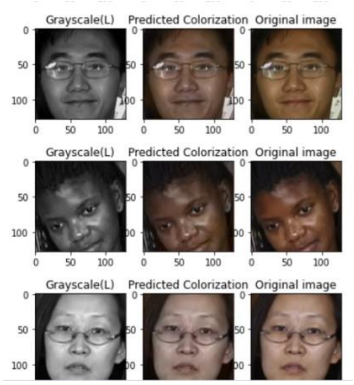


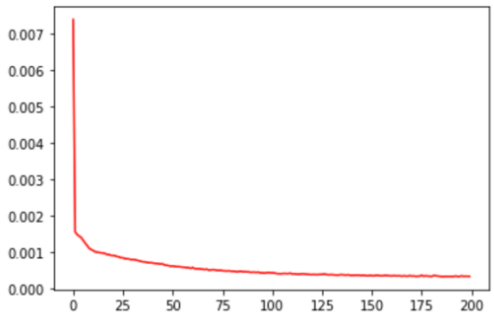
Figure 4 : The images show the results of our model on the testing set. Left image is the input L channel of the model. Middle is predicted a and b channels combined with L and right image is the ground truth RGB image.

Extra Credit :

Tanh Activation function

1. We changed our network to incorporate tanh activation function at the output layer. To get accurate results using tanh, we had to rescale the a and b channel values to lie between -1 and 1.
2. Since the values in a and b channel lie between -128 to 127, we achieve the scaling by diving each value by 128. Similarly, to rescale the values back during merging we multiply each value by 128.

Figure 5 shows the graph for training loss and the results we got on test images using the tanh function. We obtained an average loss of 0.0013. This value is very less compared to what we get without using a Tanh activation function because the error is calculated on a and b channels whose values are scaled between -1 to 1.



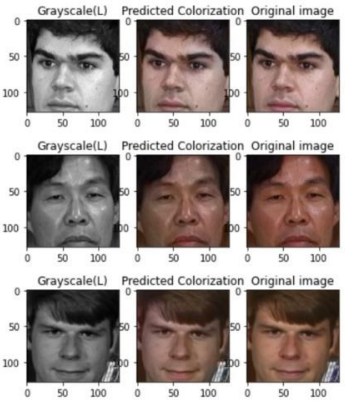


Figure 5 : Left image shows the graph for training loss for 200 epochs on the network using tanh activation function. X axis represents the epochs. Y axis represents the training loss. Right image show the results of our model with tanh activation function on the testing set. Left image is the input of the model. Middle image is the one predicted by our model and right image is the ground truth.

Experiment with different feature maps

We changed our model design to have different feature maps, to get the best network. We tried 3 different networks, feature maps for each is given in Figure 6. We received our best results from Network 3, which gave us a test error of 22.19. Last table in figure 6 also gives the average mean squared error for each network.

Downsample				Upsample	
Modules	In Channel	Out Channel	Modules	In Channel	Out Channel
Module_1	1	3	Module_6	3	3
Module_2	3	3	Module_7	3	3
Module_3	3	3	Module_8	3	3
Module_4	3	3	Module_9	3	3
Module_5	3	3	Module_10	3	2

Downsample			Upsample		
Modules	In Channel	Out Channel	Modules	In Channel	Out Channel
Module_1	1	8	Module_6	128	64
Module_2	8	16	Module_7	64	32
Module_3	16	32	Module_8	32	16
Module_4	32	64	Module_9	16	8
Module_5	64	128	Module_10	8	2

Downsample			Upsample		
Modules	In Channel	Out Channel	Modules	In Channel	Out Channel
Module_1	1	16	Module_6	256	128
Module_2	16	32	Module_7	128	64
Module_3	32	64	Module_8	64	32
Module_4	64	128	Module_9	32	16
Module_5	128	256	Module_10	16	2

Network	Test Error
1	32.89
2	23.62
3	22.19

Figure 6 : The top Left table shows how the channels vary in each module for Network 1, top right shows the channels in network 2. Bottom left shows the channel in Network 3. A single module consists of convolution or deconvolution layer, followed by batch normalization and leaky ReLU activation function (except the output layer). Bottom right shows the average mean squared error for the three networks.

Colorization using classifier

The image colorization problem involves prediction of several possible colors for single pixel. This multimodal nature of the problem is not very well suited for regression. The reason is that regression uses L2 norm to get the loss between predicted and ground truth value. This is not robust to inherent nature of the colorization problem. If, say an object could potentially take distinct set of ab values, the L2 loss will be mean of this set. This causes grayish and desaturated results.

Since the goal of the problem is to generate the possible colors, multi modal classification can be used to solve the problem. This can be done by predicting the distribution for each pixel and quantizing the ab output space into bins. As stated in the paper, further improvement with classification can be achieved through class rebalancing. Because of the points discussed above, classification approach can produce better results.