Team [40] Image Colorization using Deep Learning

1st Adya Abhyankar
aaabhyan@ncsu.edu

2nd Aadil Khan
akhan23@ncsu.edu

3rd Prathamesh Pandit
pppandi2@ncsu.edu

***Abstract*—There are several solutions available for the Image Colorization problem. A lot of research is going on in this field to obtain colorized images that are as realistic as possible. Color of objects such as Cars, Clothes etc. can be different from the ground truth but can still be considered accurate. However, the challenge lies in accurately colorizing standard components like skin-tone, eyes, hair, nature, sky etc. Our goal with this project is to produce colored images from grayscale ones that seem natural to the human eye. In our implementation, we propose a method for image colorization. We preprocess colored images to create grey-scale images which are used as input for our model. We then train our model with these grey-scale images as input and the original colored images as output. Upon training the model, we would be able to generate an RGB image with a reasonable understanding of color within its inherent texture.**

***Index Terms*—CNN, deep learning, colorization, neural networks, autoencoders, hypercolumns**

I. Model Training and Selection

A. The Model

Initially, we had planned to use image hypercolumns [3] for training our model. We implemented this but due to complexity of the parameters for training, we decided to use a simpler model to implement and use this as a baseline.

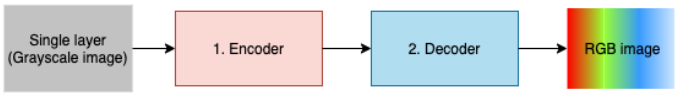
Why autoencoders? We know that autoencoders are used to reconstruct the input images in addition to its purpose of denoising. It is used in various image segmentation tasks. So the image colorization task is necessarily a part of segmentation or it involves segmentation. We cannot fill a particular part with some color unless we have segmented the picture to be made up of different objects. So, we thought that by using such a model, we would get good results in less amount of training.

Model structure: Autoencoder [2] consists of two parts: first is the encoder and second is the decoder. Encoder takes the grayscale image as its input. We converted our images to grayscale, and before that as a part of pre-processing we normalized them first. The encoder then converts the input grayscale image into a compressed form. The next part is the decoder. Its input is the output of the encoder. It takes the compressed image representation and reconstructs a RGB image from it. We trained our model on different datasets such as CIFAR and ImageNet for different image sizes. CIFAR consisted of 32x32 images. For the Imagenet dataset, we had to try different sizes to resize the images as they were very large. We tried with sizes 224x224, 128x128 and 64x64. The image with size 64x64 is what gave us the best results.

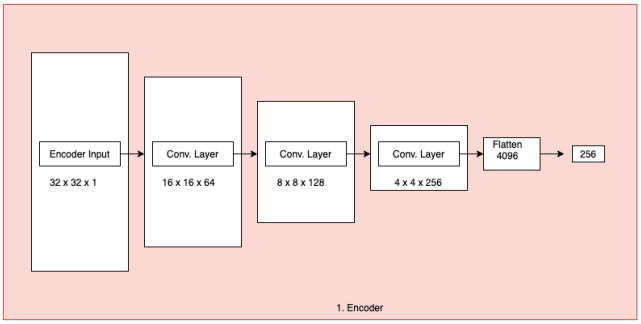
The summary of the model trained on these images is given below. The number of layers, number of parameters and the latent layer size can be seen in this summary.

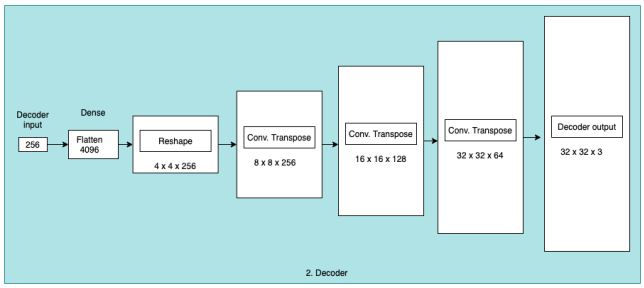
```
Model: "encoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_input (InputLayer)   [(None, 64, 64, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 32, 32, 64)        640
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 128)       73856
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 256)         295168
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 512)         1180160
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
latent_vector (Dense)        (None, 512)               4194816
=================================================================
Total params: 5,744,640
Trainable params: 5,744,640
Non-trainable params: 0
_________________________________________________________________
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
decoder_input (InputLayer)   [(None, 512)]             0
_________________________________________________________________
dense (Dense)                (None, 8192)              4202496
_________________________________________________________________
reshape (Reshape)            (None, 4, 4, 512)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 8, 8, 512)         2359808
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 16, 16, 256)       1179904
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 32, 32, 128)       295040
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 64, 64, 64)        73792
_________________________________________________________________
decoder_output (Conv2DTransp (None, 64, 64, 3)         1731
=================================================================
Total params: 8,112,771
Trainable params: 8,112,771
Non-trainable params: 0
_________________________________________________________________
Model: "autoencoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
encoder_input (InputLayer)   [(None, 64, 64, 1)]       0
_________________________________________________________________
encoder (Model)              (None, 512)               5744640
_________________________________________________________________
decoder (Model)              (None, 64, 64, 3)         8112771
=================================================================
Total params: 13,857,411
Trainable params: 13,857,411
Non-trainable params: 0
_________________________________________________________________
```

The basic autoencoder flow can be given as:



As we can see that the input is a grayscale image given to the encoder, then the output of tgis to the decoder and finally we get the RGB image at the output of the decoder. The basic architectures for the encoder and decoder can be seen below. The architecture drawn is for the model trained on CIFAR dataset containing 32x32 images.





B. Baseline

We will use the transfer learning method as the baseline. The block diagram of the approach that was used is given in Figure 1. The Implementation approach [1] can be explained as follows:
The system can be divided into two parts. These are explained below:

1) Feature-Net:
   To color an image, we first need to determine the macro colors of the image. Also, the pixels that are close to each other must have similar colors. To obtain these features, the intermediate features of a pre-trained image network VGG-16 will be used. These features will be the hypercolumns of our network. The hypercolumn of a pixel is vector of activations of all CNN units above that pixel. The local information will be obtained from the lower layers of VGG-16 while the global information will be obtained from the higher layers. This is why we extract the hyercolumns from the first 4 convolutional blocks of the VGG-16 network. These hypercolumns contain a lot of information about the image and thus it can be used for image colorization. However, as we go deeper into the VGG-16 network, due to max-pooling the size of the output of the convolutional layers starts decreasing. Due to this, the hypercolumns associated to different convolutional layers have different dimensions. For them to be used as input to some other network, it is necessary to first upscale them so that they have the same dimensions. In this case we upscale them to size (224, 224). We get a total of 960 hypercolumns so the dimensions of the final output from the VGG-16 network is (960, 224, 224). A grayscale image is given as input to the VGG-16 network and with the help of the keras backend function we extracted the hypercolumns.
2) Shallow-Net:
   The output of Feature-Net(hypercolumns) will be given to Shallow-Net which is a small CNN. This involves performing 2 3x3 convolutional operations of depth 128 and 64 successively. Since this model has to learn the UV channels of the true image we do not perform max-pooling. We train the model to learn only the UV channels and not the Y channel since the Y channel gives only the intensity of the image and can be replaced with the grayscale image. So, the grayscale image is concatenated with the UV channels to obtain the YUV image. The loss function used here is the Mean Squared error between the UV channel of the true image and the predicted image. This YUV image is then converted to an RGB image and given as the final output.

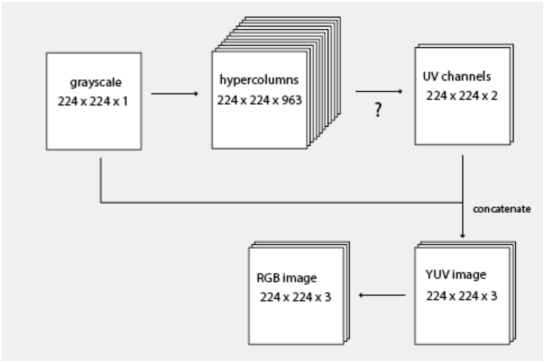


Fig. 1: Baseline model

Drawbacks:

1) The biggest drawback of this approach was it's memory requirements. Each grayscale image was converted to hypercolumns of dimensions (960, 224, 224) storing which requires a lot of memory. Infact the batch size had to be limited to only 2 images because it simply wasn't possible increase the batch size beyond that and not exhaust all the memory.
2) This model was trained for 10-12 epochs which took several hours and the model managed to learn only a brownish skin tone after this. Based on some reports, the model needs to be trained for 2-3 days on very powerful GPUs for it to be able to learn the entire color mapping.
3) Due to the long training times, hyperparameter optimization was very. difficult as any small change meant that the model had to be retrained making it very infeasible.

II. Experimental Section

A. Metrics

There are several things that we have to look for to see if our modelling is performing better. These are the things by which we can judge if the algorithm is doing a good job in colorizing the images:

- Is the color filled in proper segmented objects from a image? We have to see are the objects properly segmented and then the color given to each object.
- Is the model able to learn different colors in the image or is only learning one color for example only the brown tone in the images. This we observed in the model with hypercolumns.
- We have to look for the actual color in the original image and whether it is colored correctly in the reconstructed image. For example, if there is an image containing grass, it should be correctly colored as green in the reconstructed image.
- Is the color tone of the image appropriate?
- Is the object colored correctly in the reconstructed image for objects having multiple colors? For example a car or a bottle can have multiple colors.

B. Model selection

The primary hyperparameters for the model are:

- Batch size
- Number of epochs
- Depth of the Encoder-Decoder layers
- Kernel Size for the convolutional layers
- Filter Depth for the convolutional layers
- Size of the latent space for the Encoder-Decoder pair

Another hyperparameter that we chose implicitly was the size of the input. It was not possible to perform a grid-search over the entire space represented by the above hyperparameters. Thus, we set some hyperparameters to tried and tested values used in the industry. They are as follows:

- Batch Size = 32
- Kernel Size for the convolutional layers = 3
- Filter Depth for the layers in the Encoder-Decoder architecture: We followed the practice of keeping the filter sizes in powers of two, starting from 64 and going up to 512 for the encoder, and starting from 512 and scaling down to 64 for the decoder.

We chose the following hyperparameters based on the our observations from our experiments:

- Epochs: We set the value of epochs to 30 after observing the change in validation loss.
- Depth of Encoder-Decoder layers: we started with a 3 layer deep encoder for the CIFAR dataset, but realized that we needed greater depth for the larger imagenet images. Thus we increased the depth to 4 layer in the encoder and 4 layer in the decoder.
- Size of the latent space: Keeping the latent space size small resulted in increased blurring in the output as the data loss was not tolerated by the model. We finally settled on a latent space size of 512 for the ImageNet dataset. We found that increasing it further reduced the performance of the model.

We chose the ImageNet dataset over the Cifar10 dataset as we wanted our model to be able to learn to work on a greater range of images. Our choice of input size can be explained from the graphs below. As we can see, we get very good loss values for the Cifar10 dataset. This dataset has 10 classes and is very small (32 x 32), compared to the 100 classes we chose from the ImageNet dataset. The loss and accuracy are much better for the 64x64 resized ImageNet data classes as compared to the 128x128 dataset, and thus we decided to go ahead with the 64x64 resized ImageNet dataset for developing our final model.

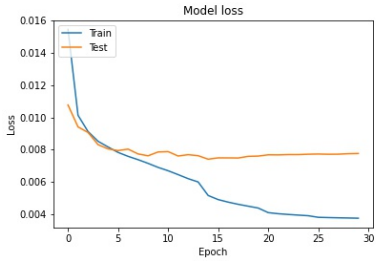


Fig. 2: CIFAR loss graph

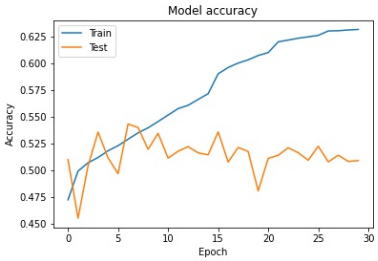


Fig. 3: CIFAR accuracy graph

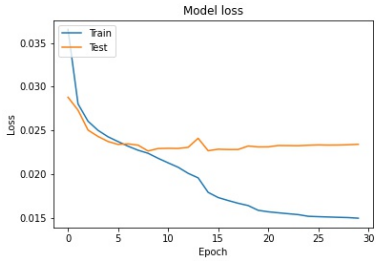


Fig. 4: ImageNet loss graph for 64x64 input size

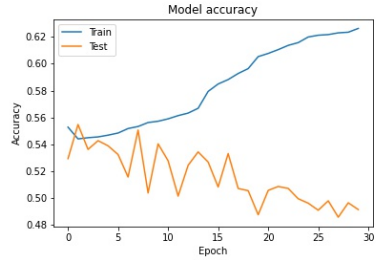


Fig. 5: ImageNet accuracy graph for 64x64 input size

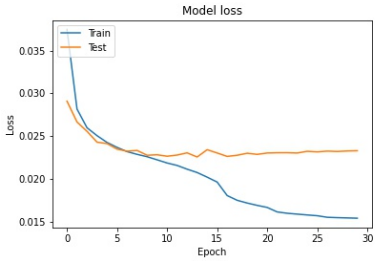


Fig. 6: ImageNet loss graph for 128x128 input size

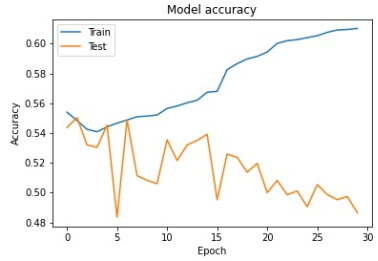


Fig. 7: ImageNet accuracy graph for 128x128 input size

C. *Performance and comparison to baseline*

As mentioned earlier, our baseline model was not memory efficient and required a lot of time to train. We could not increase our batch size above two so as to prevent memory exhaustion. We were not able to train it enough so as to capture or learn colorization properly. However upon learning for a several hours, these were the results we obtained. Figure 8 shows us the outputs generated using our baseline model. As seen from the figure, it did not provide good results as the generated image had a brownish color, mostly due to skin tone present in original image.

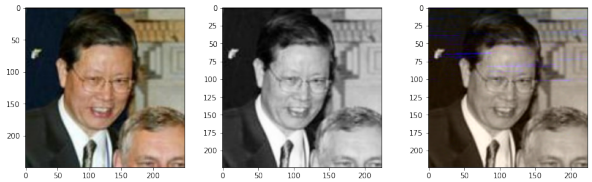


Fig. 8: Results vs Original for Baseline

We initially trained our model on 50,000 images from the CIFAR dataset. Since the images were small in size, and we made use of an Autoencoder, training did not take a lot of time. Figures 9, 10, 11 and 12 shows us the results obtained by our implementation on the model being trained and images colorized from the CIFAR dataset. As seen, these images are able to produce much better results compared to the baseline.

Thus the final model was able to perform much better than the baseline, even with less number of epochs. The colors were generated mostly accurately ignoring color of objects like the car seen in Fig 9

Upon obtaining encouraging results from training on the CIFAR dataset, we decided to train our model on 20,000 images from ImageNet. Figures 13 and 14 show the results obtained by training the model on and colorizing images from the ImageNet dataset.

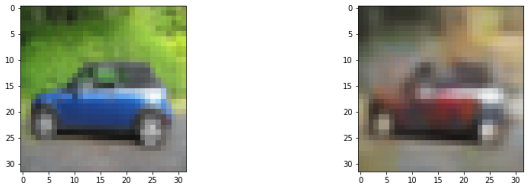


Fig. 9: Original vs Results (Cifar Dataset)

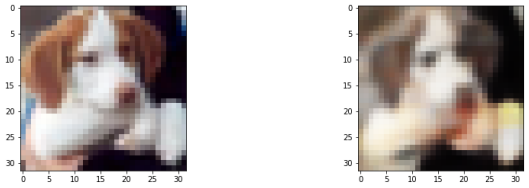


Fig. 10: Original vs Results (Cifar Dataset)

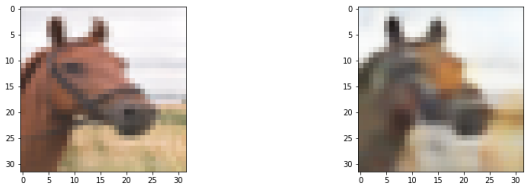


Fig. 11: Original vs Results (Cifar Dataset)

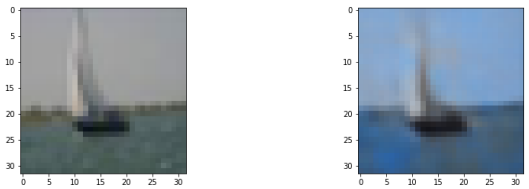


Fig. 12: Original vs Results (Cifar Dataset)

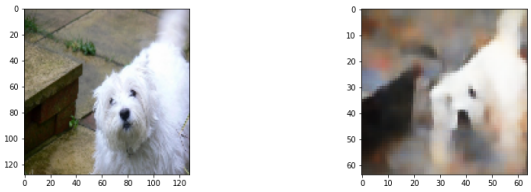


Fig. 13: Original vs Results (ImageNet Dataset)

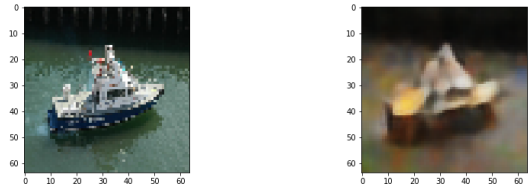


Fig. 14: Original vs Results (ImageNet Dataset)

REFERENCES

[1] Ryan Dahl. Automatic colorization.

[2] Aditya Deshpande, Jiajun Lu, Mao-Chuang Yeh, Min Jin Chong, and David Forsyth. Learning diverse image colorization. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[3] Bharath Hariharan, Pablo Arbeláez, Ross Girshick, and Jitendra Malik. Hypercolumns for object segmentation and fine-grained localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 447–456, 2015.