Image Colorization for Grayscale Images

Maharishi International University

Mario Matos, Essey Abraham Tezare, Eden Teclemariam Zere
Department of Computer Science
CS582 – Machine Learning
Spring 2020

Abstract

Deriving color from a grayscale picture can be a challenging task, but statistics and machines can do a surprisingly good job at it. Here we experiment with AutoEncoder, RestNet50, VGG6 and U-Net architectures of Convolution Neural Networks with varying results. Images for training and testing needed to be prepared to be able to train the networks correctly. Some of the models we tried gave surprising results, which we present here.

1. Introduction

Inference of color from a grayscale image is an important problem. It is bound to be imperfect without background information about the target picture for some features in the images, this is the case for clothing garments or other identifying qualities. For other common objects, their color can be ascertained by simple identification of the object, for example grass being green, sky being blue, a cow being black and white and so on. We try to implement the right neural network architecture that will be able to extract these features and infer their color appropriately to produce an image that will approximate the actual colors.

The first task to train the models is to prepare data for training and testing. We created a script to convert images to grayscale to be used as input for training and used the originals as expected output. We normalized the scale of the images to a 300x300 pixels, dropping any unnecessary edges.

```
find ${IMG_DIR} -iname '*.jpg' -exec convert \{}
-resize x600 -resize '600x<' -resize 50% -gravity
center -crop 300x300+0+0 +repage \{} \;
```

Training the models we used a split of 85% training images and 15% test images. We wanted to explore models that were more suitable for working with vision tasks. We found that AutoEncoders or architectures with a similar form worked best because we could input an image as black and white and the same image but in color as an output to correspond with this input. We hope this would make the model learn what features and other hidden patterns make color appear.

2. Exploratory Data Analysis

2.1 Dataset Features

One of our datasets used was the COCO[1] dataset with varied images of different objects, scenery and overall environments, which made it suitable for our experiments.

To normalize the images we resized them to 300x300 pixels, centered them in an attempt to capture the most important part of every image, converted to LAB format which has a component of lightness (similar to grayscale image), a blue-green gradient component and a red-yellow component.

2.2 Transfer Learning

We wanted to explore the idea that a network that is capable of identifying objects could be further trained to infer the coloration of such objects[2]. Therefore, we included the 'imagenet' dataset in some of our models to compare results with models that didn't have any pre-trained layers.

2.2 Color Manipulation

As we mentioned in the sections above, we used the LAB colormap because it could be more suitable to train our model with more clear targets [3]. This however came at the cost of converting all images to LAB for processing. Also, we used the reverse process to convert our predicted images from LAB back to RGB.
To help with conversion we used the scikit-images library and applied the following operation for normalization:

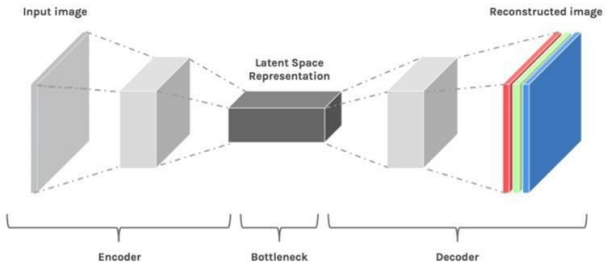
```
def normalize_lab(img):
        return (img + [0, 128, 128])/[100, 255, 255]
def denormalize_lab(img):
        return (img * [100, 255, 255]) - [0, 128, 128]
def convert_rgb_to_lab(img):
        return normalize_lab(rgb2lab(img))
def convert_lab_to_rgb(img):
        return lab2rgb(denormalize_lab(img))
```

Example image from COCO dataset.

Before applying the normalization for RGB images, all values are divided by 255 to scale the inputs appropriately. After applying the normalization for LAB the values are in turn multiplied by 255, this of course assumes that the original images have color values that range 0-255, if not rescale as needed.

3. Methods

In some capacity the models that can produce an image with color from a grayscale one have an architecture similar to AutoEncoder. So let us discuss this model first. Our input shape is (N, 300, 300, 1) and output shape (N, 300, 300, 2) in terms of numpy arrays. This allows us to use the lightness dimension of the LAB formatted image as input and the color layers (a*b) as output for training. An approach tried was using a pre trained layer right after the input layer to be able to try to have the learned objects that are part of imagenet dataset (ResNet50 with 'imagenet' for weights hyperparameter). Another approach was the plain AutoEncoder architecture where features are reduced by convoluting and therefore hopefully reveal the latent features. For these we used 2500 images with epochs of 50 – 500.



Auto Encoder Architecture [4]

Another network we tried was using VGG16 with epochs of 1000 on a dataset of ~63 thousand images. For this experiment however, instead of using the L component

of LAB we repeated this component in the 3 dimensions of the input image to create the grayscale. This produced good results even having good performance in spite of having more input dimensions.

Another network architecture we tried was U-Net. U-Net in our opinion also has similar architecture to AutoEncoders, however U-Net has more latent feature spaces and also skip connections are used to transfer information from lower levels to higher levels[5].

4. Experiments

Plain old AutoEncoder :

```
model = Sequential()
model.add(Conv2D(64, (3, 3), activation='relu', padding='same',
strides=2, input_shape=(256, 256, 1)))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(128, (3,3), activation='relu', padding='same',
strides=2))
model.add(Conv2D(256, (3,3), activation='relu', padding='same'))
model.add(Conv2D(256, (3,3), activation='relu', padding='same',
strides=2))
model.add(Conv2D(512, (3,3), activation='relu', padding='same'))
model.add(Conv2D(512, (3,3), activation='relu', padding='same'))
model.add(Conv2D(256, (3,3), activation='relu', padding='same'))
model.add(Conv2D(128, (3,3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3,3), activation='relu', padding='same'))
model.add(Conv2D(16, (3,3), activation='relu', padding='same'))
model.add(Conv2D(2, (3, 3), activation='tanh', padding='same'))
model.add(UpSampling2D((2, 2)))

model.compile(optimizer='adam', loss='mse' , metrics=['accuracy'])
history=model.fit(X,Y,validation_split=0.1, epochs=500, batch_size=16)
```

AutoEncoder with embedded pretrained imagenet frozen layer:

```
model = Sequential()
model.add(InputLayer(input_shape=(IMG_WIDTH, IMG_HEIGHT, 1)))
model.add(UpSampling3D((2, 2, 3)))
model.add(ResNet50(weights='imagenet', include_top=False))
model.add(UpSampling2D((4, 4)))
model.layers[-1].trainable = False
model.add(Conv2D(8, (4, 4), activation='relu', padding='valid', strides=2))
model.add(Conv2D(16, (4, 4), activation='relu', padding='same'))
model.add(Conv2D(16, (3, 3), activation='relu', padding='same', strides=2))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', strides=2))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
model.add(UpSampling2D((3, 3)))
model.add(Conv2D(16, (2, 2), activation='relu', padding='same'))
model.add(UpSampling2D((5, 5)))
model.add(Conv2D(2, (3, 3), activation='tanh', padding='same'))
model.summary()

model.compile(optimizer='rmsprop', loss='mse', metrics=['accuracy', 'val_accuracy'])
model.fit(x=train_X, y=train_y, batch_size=5, epochs=500, verbose=1)
```

VGG16 we used 19 layers of pre-trained

```
vggmodel = VGG16()
newmodel = Sequential()
for i, layer in enumerate(vggmodel.layers):
    if i<19:                    newmodel.add(layer)
for layer in newmodel.layers:
    layer.trainable=False


vggfeatures = []
for i, sample in enumerate(X):
    sample = gray2rgb(sample)
    sample = sample.reshape((1,224,224,3))
    prediction = newmodel.predict(sample)
    prediction = prediction.reshape((7,7,512))
    vggfeatures.append(prediction)
vggfeatures = np.array(vggfeatures)

model = Sequential()

model.add(Conv2D(256, (3,3), activation='relu', padding='same',
input_shape=(7,7,512)))
model.add(Conv2D(128, (3,3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(64, (3,3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(32, (3,3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(16, (3,3), activation='relu', padding='same'))
model.add(UpSampling2D((2, 2)))
model.add(Conv2D(2, (3, 3), activation='tanh', padding='same'))
model.add(UpSampling2D((2, 2)))

model.compile(optimizer='Adam', loss='mse' , metrics=['accuracy'])
history=model.fit(vggfeatures, Y,validation_split=0.1 ,verbose=1,
epochs=1000, batch_size=128)
```

Trying U-Net as well[7] same implementation as cited, but with our own (COCO) dataset which is larger and has images of larger dimensions:

```
def conv2d(layer_input, filters, f_size=4, bn=True):
    """Layers used during downsampling"""
    d = Conv2D(filters, kernel_size=f_size, strides=2, padding='same')(layer_input)
    d = LeakyReLU(alpha=0.2)(d)
    if bn:
        d = BatchNormalization(momentum=0.8)(d)
    return d

def deconv2d(layer_input, skip_input, filters, f_size=4, dropout_rate=0):
    """Layers used during upsampling"""
    u = UpSampling2D(size=2)(layer_input)
    u = Conv2D(filters, kernel_size=f_size, strides=1, padding='same',
activation='relu')(u)
    if dropout_rate:
        u = Dropout(dropout_rate)(u)
    u = BatchNormalization(momentum=0.8)(u)
    u = Concatenate()([u, skip_input])
    return u

gf = 32 # number of filters
# Image input
d0 = Input(shape=(256, 256, 1))

# Downsampling
d1 = conv2d(d0, gf)
d2 = conv2d(d1, gf*2)
d3 = conv2d(d2, gf*4)
d4 = conv2d(d3, gf*8)
d5 = conv2d(d4, gf*8)
d6 = conv2d(d5, gf*8)
d7 = conv2d(d6, gf*8)
```

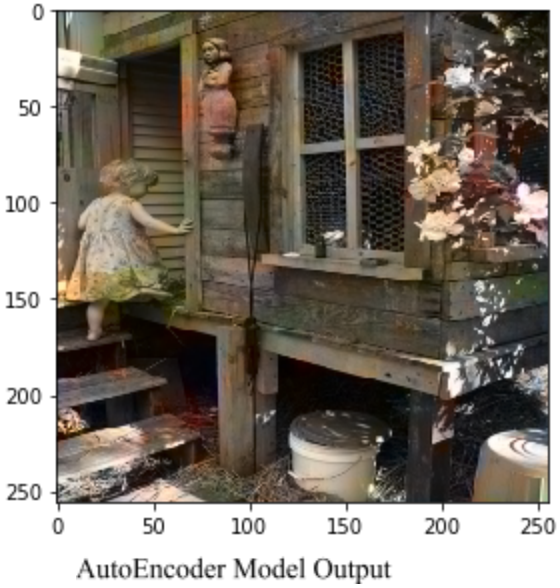
```
# Upsampling
u1 = deconv2d(d7, d6, gf*8)
u2 = deconv2d(u1, d5, gf*8)
u3 = deconv2d(u2, d4, gf*8)
u4 = deconv2d(u3, d3, gf*4)
u5 = deconv2d(u4, d2, gf*2)
u6 = deconv2d(u5, d1, gf)

u7 = UpSampling2D(size=2)(u6)
output_img = Conv2D(3, kernel_size=4, strides=1, padding='same',
activation='sigmoid')(u7)

model = Model(d0, output_img)
model.compile(optimizer='adam', loss='mse', metrics=['mse', 'mae'])
model.fit_generator(train_generator, steps_per_epoch=20, epochs=2,
validation_data=validation_generator, validation_steps=3)
```

Also we decided to try an AutoEncoder using our own dataset[6][7]

5. Results

Input RGB Image

Grayscale Image



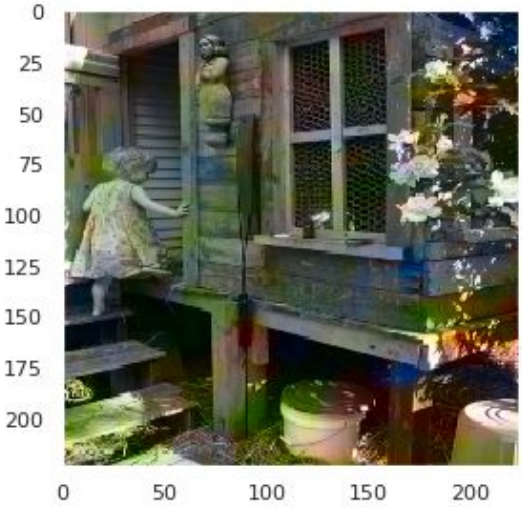
AutoEncoder Model Output

The best predicted colors were achieved using the AutoEncoder model. Trained with 150 epochs, batch size of 16, on a dataset comprising approximately 63 thousand images. With this model the best validation accuracy we got as around 0.60.

Several of the images we handpicked for visual inspection had coloration over inserted by our model which at first inspection looks realistic.

The AutoEncoder with pretrained imagenet dataset frozen produced somewhat good results but required long time for training due to the number of parameters that are connected to the layers ready to be trained. Epochs of 50 and limiting data to 500 images produced images with shades of coloring that could subjectively be cataloged as correctly placed.

Our implementation of VGG16 got similar results to the AutoEncoder, with the added bonus of a huge boost in performance while training. We believe that this improvement of performance over our other experiment may be due to the simplicity in design and therefore less complex flow of information through the network. Validation accuracy was 0.50, which considering the task at hand it is worth appraising as good.



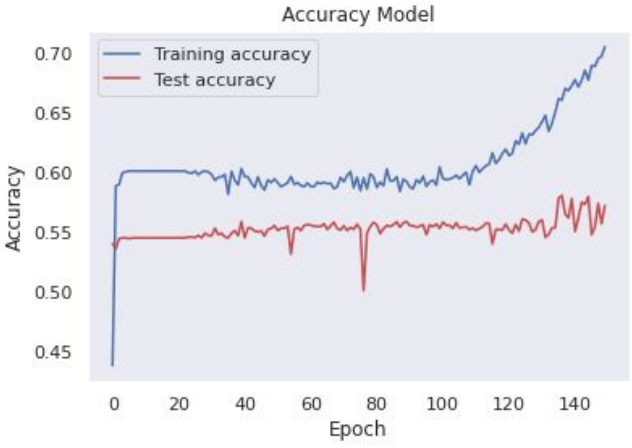
VGG16 Predicted Colors

The U-Net and last AutoEncoder we tried compare to our results closely.

We had very good results with the U-Net, however the first AutoEncoder model (with sample picture in this document) did considerably better.

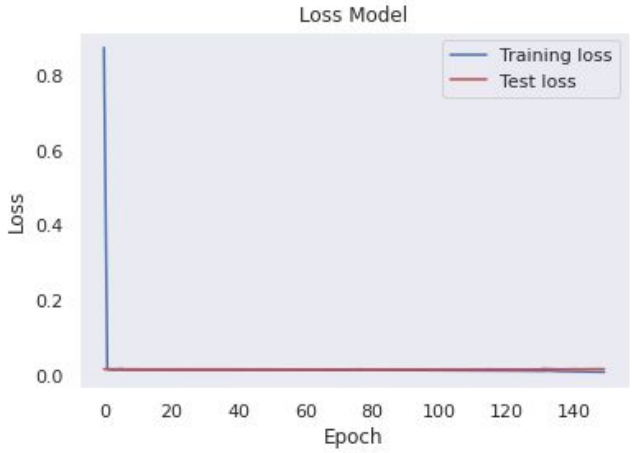
One of the most interesting experiments did was with the last AutoEncoder. At first, the output was just a grayscale image with no color. We decided to increase the epochs for training which confusingly resulted in images having over coloration and being completely green. After thinking about what the problem might be we thought of overfitting and also noticed that the validation curves showed learning and validation being off at certain number of training epochs. So, to use all the training we had and have meaningful number of epochs, a dropout[9] layer was introduced to prolong the time spent learning.

5.1 Learning and Validation Curves

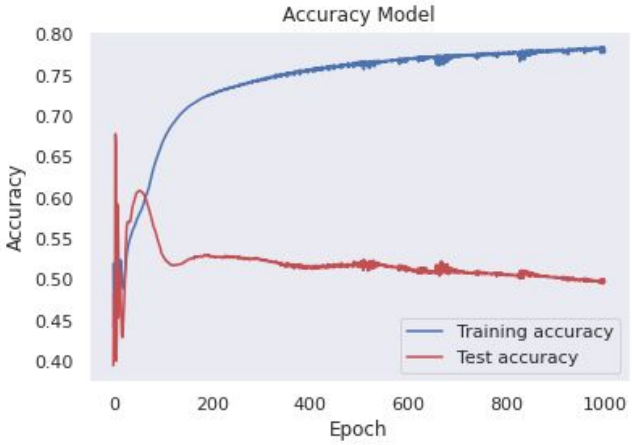


AutoEncoder that produced good results had the following learning curve and loss functions:

We can see that the gain in accuracy for both training and testing flattens out quickly improving after around 120 epochs.

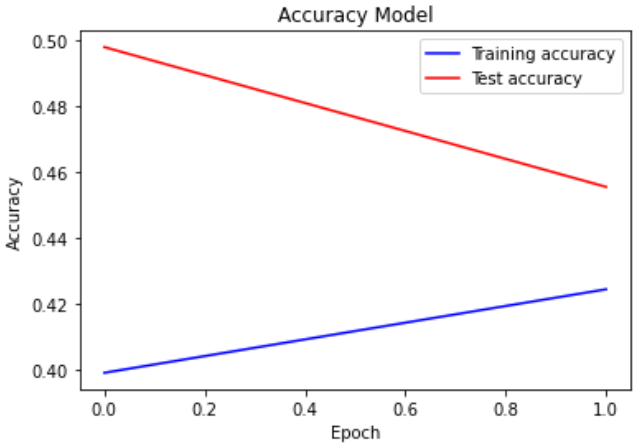


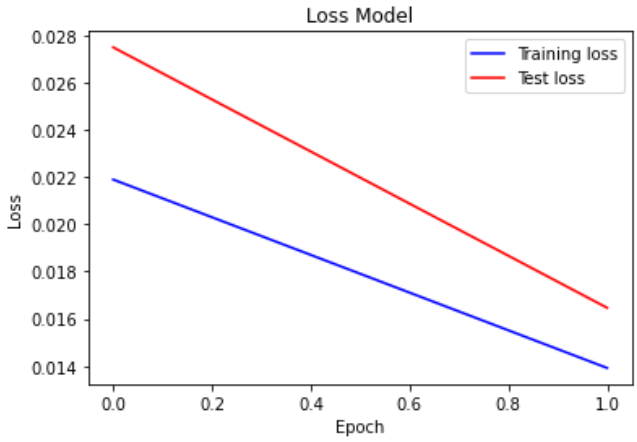
For the VGG16 model:



Testing and training accuracy for the VGG16 shown in the graph above intercept at around 100 epochs. So this should be our target for optimal performance.

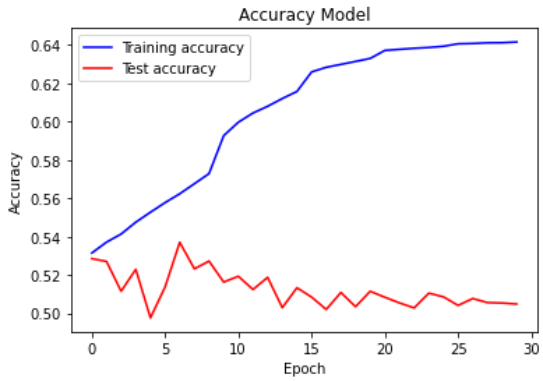
U-Net:

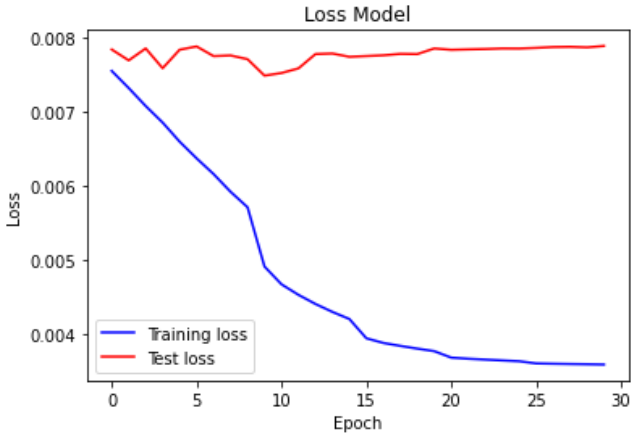




With more training for the U-Net model the testing and training accuracy should intercept. Despite this, this architecture yielded somewhat good results as designed by its implementer.

Second AutoEncoder we tried with our data:





6. Conclusion

After experimenting on this problem domain with various architectures, it is apparent that the simpler the better.

Trying imagenet as a pre-trained ResNet50 layer was harder to train because all the pretrained weights take time to update (32 million weights) but it yielded good results while training for a lower amount of epochs.

The AutoEncoder with its common plain design of convolutions and upscalers seem to be the best arrangement to produce color in grayscale images and with moderate performance. VGG seems to be an improvement in training time since epochs of training would flow fast, but with the added cost of less quality prediction.

7. Contributors

- Mario Matos
- Essey Abraham Tezare
- Eden Teclemariam Zere

References

[1] Lin, T., Patterson, G., et al (2017). Common Objects in Context. Retrieved July 14, 2020, from https://cocodataset.org/

[2] Gowda, S. N., & Yuan, C. (2019). ColorNet: Investigating the Importance of Color Spaces for Image Classification. *Computer Vision – ACCV 2018 Lecture Notes in Computer Science,* 581-596. doi:10.1007/978-3-030-20870-7_36

[3] Identifying Color Differences Using L*a*b* or L*C*H* Coordinates. (2020, February 07). Retrieved July 16, 2020, from https://sensing.konicaminolta.us/us/blog/identifying-color-differences-using-l-a-b-or-l-c-h-coordinates/

[4] Patel, H. (2020, June 15). Image Super-Resolution using Convolution Neural Networks and Auto-encoders. Retrieved July 16, 2020, from https://towardsdatascience.com/image-super-resolution-using-convolution-neural-networks-and-auto-encoders-28c9eceadf90

[5] Kizrak, A. (2020, January 29). Deep Learning for Image Segmentation: U-Net Architecture. Retrieved July 16, 2020, from https://heartbeat.fritz.ai/deep-learning-for-image-segmentation-u-net-architecture-ff17f6e4c1cf

[6] Atienza, R. (2010, December 8). PacktPublishing/Advanced-Deep-Learning-with-Keras. Retrieved July 16, 2020, from https://github.com/PacktPublishing/Advanced-Deep-Learning-with-Keras/blob/master/chapter3-autoencoders/colorization-autoencoder-cifar10-3.4.1.py

[7] Bhattiprolu, S. (2020, July 16). Python for Microscopists. Retrieved July 16, 2020, from https://github.com/bnsreenu/python_for_microscopists

[8] Hsankesara. (2018, June 12). Flickr Image dataset. Retrieved July 16, 2020, from https://www.kaggle.com/hsankesara/flickr-image-dataset

[9] Brownlee, J. (2019, September 13). Dropout Regularization in Deep Learning Models With Keras. Retrieved July 16, 2020, from https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/