Converting Grayscale Images to RGB Images Using Various Autoencoder Architectures

Christian Durant, Abhinav Kotta, Jackson Mazer

January 29, 2024

Abstract

In this project, we designed and experimented with various autoencoder architectures aimed toward converting grayscale images into their RGB counterparts. We utilized the 32x32 RGB images of the CIFAR10 dataset to test the models' conversion of their grayscale versions into their original RGB formats. The focus of the experimentation was to create a versatile model capable of generating RGB images from newly seen grayscale images and to identify the components of an autoencoder architecture that produce the best results. We tested a 3-layer convolutional model, a linear latent model, and a CNN latent model, and determined the 3-layer convolutional model produced the superior reconstructed colorized images.

1 Introduction

Throughout history, people have created brilliantly-colored architecture, clothing, paintings, and more. Any medium in which human creativity could be expressed witnessed an explosion of color. While we see their architecture as stark white in modern times, we know the ancient Romans bathed their statues, temples, and homes in splashes of bright colored paints, for example. Art movements throughout the centuries have always brought with them new trends and discoveries in color theory and creativity. Photography, then, became the next medium for popular colorful expression.

Color photography has been possible for over one hundred years, and has been widely found throughout the world for over fifty. Starting in the mid-1960s, the advent of widely-used color camera technology was brought to the business market, and in 1990, the first digital color camera was brought to the consumer market. Since then, camera technology has only improved to the point that the average person has a 12–48 megapixel camera in their pocket. But with such widely accessible high-quality color photography, the desire to colorize photos from before this time has become that much more popular. This desire is achievable with an autoencoder, and it is this desire that we sought to fulfill with this project.

2 Method

An autoencoder, or convolutional encoder-decoder model, is a type of layered convolutional neural network designed to compress and reconstruct an image. In doing so, the autoencoder seeks to predict the same image as an output as what it received as an input. As opposed to the typical one dataset input used with a convolutional neural network, we use two datasets when colorizing grayscale images. By presenting the encoder with colored images to learn the features from, and only then giving it grayscale images to reconstruct, the decoder will build back to the images using the color features learned initially. In this way, the autoencoder will break down a black-and-white image and build up a color image.

As a team, we decided to each develop a slightly different architecture for the encoder and decoder and each test them independently using the same dataset. Once that was completed, we would compare and contrast the results from each architecture and determine which was best for reconstruction.

3 Experiments

3.1 Dataset

The dataset we used for the autoencoder model was the CIFAR-10 dataset. This is a dataset composed of sixty-thousand 32x32 colored images separated into ten classes of five thousand training and one thousand test images each. It would make for a sufficient and easily-accessible dataset that isn't so big that training would take longer than necessary.

3.2 Evaluation metrics

For each of the models that we trained and employed, we used the mean-squared error loss function to evaluate the disparity between the autoencoders' output RGB images generated from their grayscale inputs and the original RGB images from which the grayscale images were derived. This method of evaluation allowed us to assess the autoencoders' abilities to generate the correct proportions and intensities of the three color channels of the outputs. We also took a more visual approach to evaluating the performance of the models by performing a direct comparison between the original RGB images and the output RGB images as shown below.

3.3 Results

One of the implementations of the autoencoder was an implementation using three convolutional layers as well as a dense layer in each of the encoder and decoder in Keras. The convolutional layers contain 64, 128, and 256 nodes respectively. This is a fairly standard implementation, and it produced quite good results with minimal loss and maximal accuracy.

Below are some images from the CIFAR-10 dataset next to the resultant colorized images produced by the autoencoder.



Figure 1: Images from the RGB test dataset, the grayscale test dataset, and the colorized results, respectively from the original model.



Figure 2: Output from the autoencoder architecture without a dynamically changing learning rate.

Further implementations of the autoencoder model attempted to identify the potential difference in performance between a fully connected (Dense) latent representation and a convolutional latent representation within the autoencoder architecture.

Two models were employed for such testing. One model had a fully-connected latent representation of a Linear layer consisting of 256 output nodes, while the other model had a fully convolutional latent representation consisting of 256 3x3 filters. For the fully connected latent representation, a second layer was required to upscale back to the number of neurons required for convolutional computation. Below are the outputs of two autoencoder models, implemented in PyTorch.



Figure 3: Test input and colorized output from the autoencoder architecture containing a fullyconnected latent representation.



Figure 4: Test input and colorized output from the autoencoder architecture containing a CNN representation.

3.4 Analysis and discussions

Upon examining the results of all of the models, it became more apparent what components of an autoencoder's architecture are most necessary to transform grayscale images into RGB format. The convolutional latent representation performed much better than the fully connected latent representation and also produced clearer images than that of the original configuration (Figure 1), as well. The original configuration, consisting of both convolutional layers and dense layers (similar to model 2), did manage to capture more color than the other two models, however.

The following analysis describes in detail each of the architectures that were employed throughout the experimentation as well as the change in the loss values during training. We can examine this information to determine each of the model's efficiency in learning to take the input grayscale images and generate their RGB counterparts.

3.4.1 Model Architecture and Training

Model: "encoder"			
Layer (type)	Output Shape	Param #	
encoder_input (InputLayer) conv2d (Conv2D) conv2d_1 (Conv2D) conv2d_2 (Conv2D) flatten (Flatten) lattent_vector (Dense)	[(None, 32, 32, 1)] (None, 16, 16, 64) (None, 8, 8, 128) (None, 4, 4, 256) (None, 4096) (None, 256)	0 640 73856 295168 0 1048832	
Total params: 1418496 (5.41 MB) Trainable params: 1418496 (5.41 MB) Non-trainable params: 0 (0.00 Byte)			
Model: "decoder"			
Layer (type)	Output Shape	Param #	
decoder_input (InputLayer) dense (Dense) reshape (Reshape) conv2d_transpose (Conv2DTr anspose)	[(None, 256)] (None, 4096) (None, 4, 4, 256) (None, 8, 8, 256)	0 1052672 0 590080	
<pre>conv2d_transpose_1 (Conv2D Transpose) conv2d_transpose_2 (Conv2D Transpose_2)</pre>	(None, 16, 16, 128) (None, 32, 32, 64)	295040 73792	
decoder_output (Conv2DTran spose)	(None, 32, 32, 3)	1731	
Total params: 2013315 (7.68 MB) Trainable params: 2013315 (7.68 MB) Non-trainable params: 0 (0.00 Byte)			

Figure 5: First implementation of the autoencoder using three convolutional layers and a dense layer in both the encoder and the decoder.



Figure 6: Loss values of the first model over the 30 epochs for which it was trained.



Figure 7: Loss values of the first model over the 30 epochs for which it was trained with a static learning rate.

-Conv2d: 1-1	[-1, 64, 32, 32]	640		
Conv2d: 1-2	[-1, 128, 32, 32]	73,856		
MaxPool2d: 1-3	[-1, 128, 16, 16]			
-Conv2d: 1-4	[-1, 256, 16, 16]	295,168		
MaxPool2d: 1-5	[-1, 256, 8, 8]			
-Linear: 1-6	[-1, 256]	4,194,560		
-Linear: 1-7	[-1, 16384]	4,210,688		
Upsample: 1-8	[-1, 256, 16, 16]			
-ConvTranspose2d: 1-9	[-1, 128, 16, 16]	295,040		
Upsample: 1-10	[-1, 128, 32, 32]			
-ConvTranspose2d: 1-11	[-1, 64, 32, 32]	73,792		
ConvTranspose2d: 1-12	[-1, 3, 32, 32]	1,731		
Total params: 9,145,475				
Trainable params: 9,145,475				
Non-trainable params: 0				
Total mult-adds (M): 312.74				
Input size (MB): 0.00				
Forward/backward pass size (MB): 2.90				
Params size (MB): 34.89				
Estimated Total Size (MB): 37.79				

Figure 8: Architecture of the autoencoder model containing a Linear latent representation.



Figure 9: Loss values of the FC latent space model over the 20 epochs for which it was trained.

Layer (type:depth-idx)	Output Shape	Param #		
-Conv2d: 1-1	[-1, 64, 32, 32]	640		
Conv2d: 1-2	[-1, 128, 32, 32]	73,856		
MaxPool2d: 1-3	[-1, 128, 16, 16]			
Conv2d: 1-4	[-1, 256, 16, 16]	295,168		
Conv2d: 1-5	[-1, 256, 16, 16]	590,080		
ConvTranspose2d: 1-6	[-1, 128, 16, 16]	295,040		
Upsample: 1-7	[-1, 128, 32, 32]			
ConvTranspose2d: 1-8	[-1, 64, 32, 32]	73,792		
ConvTranspose2d: 1-9	[-1, 3, 32, 32]	1,731		
Total params: 1,330,307				
Trainable params: 1,330,307				
Non-trainable params: 0				
Total mult-adds (M): 455.34				
Input size (MB): 0.00				
Forward/backward pass size (MB): 3.27				
Params size (MB): 5.07				
Estimated Total Size (MB): 8.35				

Figure 10: Architecture of the autoencoder model containing a convolutional latent representation.



Figure 11: Loss values of the CNN latent space model over the 20 epochs for which it was trained.

As shown in Figures 5 and 6, the first architecture was capable of achieving a minimum loss on the training set of approximately 0.004, allowing it to generate the RGB images as shown in the results.

3.4.2 Comparisons

Figures 1-4 display the difference between having n fully connected and convolutional latent space representation within the autoencoder. The CNN architecture managed to achieve a minimum loss comparable to that of the first model at approximately 0.005, while the FC architecture struggled to even get below 0.01. The implications of this difference are demonstrated in the output, where the output of the model with the convolutional latent space is much more defined and clear than that of the model with the linear latent space. Additionally, the vibrancy of the images reconstructed using the model with the three convolutional layers is significantly more notable than their counterparts in the other architectures.

It is also worth noting that the model containing the fully-connected latent space had significantly more parameters than the other models that were employed, causing it to take more time to train during experimentation. The model with the convolutional latent space, on the other hand, had even fewer parameters than the first model but still achieved a comparable loss, suggesting that further modification to the architecture could yield similar or better results on the test set with a fully convolutional model.

Overall, it's clear that the 3-convolutional layer architecture produced the best-looking results. While its loss was comparable to that of the CNN latent space model, both at the 20 epoch mark and what could be extrapolated to be the loss beyond that mark for the latter, the apparent reconstruction was leagues ahead in the former. The performance of the linear latent representation architecture, then, is insufficient in both categories.

3.4.3 Discussion

The experimentation with 3-layer autoencoders, fully connected (Dense) latent, and convolutional representation of models yielded significant insights into its performance and adaptability in grayscale image colorization.

The observation that a static learning rate showed a slight degradation in the colorization quality underlines the importance of adaptive learning strategies. Dynamically adjusting the learning rate when the loss becomes stationary for a certain number of epochs seems crucial for maintaining optimal convergence and enhancing the intricacies of colorized outputs, although makes a minimal difference when compared to other architectures.

The comparison between fully connected and convolutional latent spaces highlighted the superiority of convolutional architectures for capturing intricate spatial information. The convolutional latent space exhibited a more effective representation of image features by increasing the image quality compared to the fully connected latent representation.

Despite variations in architecture and latent space representation, the robustness and performance of the conventional 3-layer autoencoder stood out. Its popularity stems from its ability to strike a balance between model complexity and efficiency, offering a favorable trade-off for grayscale image colorization tasks. The adaptability and reliability of this architecture in consistently producing highquality colorized outputs make it a preferred choice among the experimented configurations.

4 Conclusion

4.1 Description

In this work, we investigate the viability of multiple different autoencoder architectures for the colorization of grayscale images. We have tested a 3-layer model with and without modifications to its learning rate, and two latent space models, both linear and CNN-composed. While the fixed learning rate slightly impacted colorization quality, the comparative analysis underscored the effectiveness of the 3-layer autoencoder architecture. The superior performance of the convolutional latent space reaffirmed its significance in preserving spatial information crucial for accurate colorization. Future research focusing on hybrid architectures integrating dynamic learning rates with convolutional latent spaces holds promise for further enhancing colorization fidelity. These insights serve as valuable guidelines for optimizing autoencoder architectures for grayscale image colorization tasks, paving the way for more refined and realistic colorization techniques in the future.

4.2 What I Learned

Throughout this endeavor, the journey of implementing the 3-layer convolutional autoencoder served as a profound learning experience. The process illuminated the significance of hands-on exploration in unfamiliar domains. I gained a comprehensive understanding of autoencoder architectures and their application in grayscale image colorization, delving into the intricate workings of TensorFlow and Keras along the way.

The introduction of a fixed learning rate into the model shed light on the nuanced interplay between training strategies and resulting image quality, underscoring the importance of adaptive learning mechanisms in optimizing model performance. This journey not only deepened my technical proficiency but also honed my ability to contribute meaningfully to collaborative research efforts, synthesizing findings and insights to inform the conclusions and future directions of our project. Overall, this experience reinforced the value of curiosity-driven learning and collaborative teamwork in navigating complex technical landscapes.

5 Contribution

5.1 Code

Below is the code for the different architectures examined in this work.

5.1.1 Three Convolutional Layer Implementation With Learning Rate Modification

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import os
4 from keras.layers import Dense, Input
5 from keras.layers import Conv2D, Flatten
6 from keras.layers import Reshape, Conv2DTranspose
```

```
7 from keras.models import Model
8 from keras.callbacks import ReduceLROnPlateau
9 from keras.callbacks import ModelCheckpoint
10\ {\rm from\ keras.datasets\ import\ cifar10}
11 from keras import backend as K
12
13 ,----,
14 'PREPARING THE DATASETS'
15 '----'
16 # Load in the images from the cifar10 dataset.
17 (x_train, _), (x_test, _) = cifar10.load_data()
18
19 # Turn the images in the test dataset to grayscale for a new test dataset
20 x_train_gray = np.dot(x_train[...,:3], [0.299, 0.587, 0.114])
21 \text{ x_test_gray} = \text{np.dot}(\text{x_test}[...,:3], [0.299, 0.587, 0.114])
22
23 # Get the size of the images
24 img_rows = x_train.shape[1]
25 img_cols = x_train.shape[2]
26 channels = x_train.shape[3]
27
28 ,----,
29 'PLOTTING THE DATASETS'
30 ,----,
31 # Store 100 images of the color test dataset
32 \text{ datasetC} = x_\text{test}[:100]
33 datasetC = datasetC.reshape((10, 10, img_rows, img_cols, channels))
34 datasetC = np.vstack([np.hstack(i) for i in datasetC])
35
36 # Plot 100 images of the color test dataset
37 plt.figure()
38 plt.axis('off')
39 plt.title('Test Images (Color)')
40 plt.imshow(datasetC, interpolation='none')
41 plt.show()
42
43 # Store 100 images of the grayscale test dataset
44 datasetG = x_test_gray[:100]
45 datasetG = datasetG.reshape((10, 10, img_rows, img_cols))
46 datasetG = np.vstack([np.hstack(i) for i in datasetG])
47
48 # Plot 100 images of the grayscale test dataset
49 plt.figure()
50 plt.axis('off')
51 plt.title('Test Images (Grayscale)')
52 plt.imshow(datasetG, interpolation='none', cmap='gray')
53 plt.show()
54
55 ,----,
56 'PREPARING FOR MODEL INPUT/OUTPUT'
57 ,----,
58 # Normalize and reshape the color datasets
59 x_train = x_train.astype('float32') / 255
60 \text{ x_test} = \text{x_test.astype('float32')} / 255
61 x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, channels)
62 x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, channels)
63
64 # Normalize and reshape the grayscale datasets
65 x_train_gray = x_train_gray.astype('float32') / 255
66 x_test_gray = x_test_gray.astype('float32') / 255
67 x_train_gray = x_train_gray.reshape(x_train_gray.shape[0], img_rows, img_cols, 1)
68 x_test_gray = x_test_gray.reshape(x_test_gray.shape[0], img_rows, img_cols, 1)
69
70 # Define parameters
71 input_shape = (img_rows, img_cols, 1)
72 \text{ batch_size} = 32
73 \text{ kernel_size} = 3
74 \text{ latent_dim} = 256
75 \text{ layer_filters} = [64, 128, 256]
76
77 '-----'
```

```
78 'BUILDING THE ENCODER MODEL'
 79 , _____,
 80
 81 # Build the model
 82 inputs = Input(shape=input_shape, name='encoder_input')
 83 x = inputs
 84
 85 # 3 Convolution layers: 64 filters, 128 filters, 256 filters
 86 for filters in layer_filters:
 87
       x = Conv2D(filters=filters,
 88
                   kernel_size=kernel_size,
 89
                   strides=2,
                   activation='relu',
 90
                   padding='same')(x)
 91
92
 93 # Reshape to 4x4x256 for processing, to eventually be shaped back into 32x32x3
94 shape = K.int_shape(x)
95
 96 # Thank you for resources online on how to do this type of thing.
97 x = Flatten()(x)
98 latent = Dense(latent_dim, name='latent_vector')(x)
99
100 encoder = Model(inputs, latent, name='encoder')
101 encoder.summary()
102
103 '----'
104 'building the decoder model'
105 '-----'
106 latent_inputs = Input(shape=(latent_dim,), name='decoder_input')
107 x = Dense(shape[1]*shape[2]*shape[3])(latent_inputs)
108 x = \text{Reshape}((\text{shape}[1], \text{shape}[2], \text{shape}[3]))(x)
109
110 # Three convolution layers: 256 filters, 128 filters, 64 filters
111 for filters in layer_filters[::-1]:
112
       x = Conv2DTranspose(filters=filters,
113
                            kernel_size=kernel_size,
114
                            strides=2.
115
                            activation='relu',
116
                            padding='same')(x)
117
118 outputs = Conv2DTranspose(filters=channels,
119
                              kernel_size=kernel_size,
120
                              activation='sigmoid',
121
                              padding='same',
122
                              name='decoder_output')(x)
123
124 decoder = Model(latent_inputs, outputs, name='decoder')
125 decoder.summary()
126
127 '-----'
128\ {\rm ,THE}\ {\rm AUTOENCODER}\,{\rm ,}
129 ,----,
130 autoencoder = Model(inputs, decoder(encoder(inputs)), name='autoencoder')
131 autoencoder.summary()
132
133 # Save as a new model any time the loss improves.
134 save_dir = os.path.join(os.getcwd(), 'saved_models')
135 model_name = 'Model{epoch:02d}.h5'
136 if not os.path.isdir(save_dir):
137
           os.makedirs(save_dir)
138 filepath = os.path.join(save_dir, model_name)
139
140 # Reduce learning rate by sqrt(0.1) if the loss does not improve in 5 epochs
141 lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
142
                                    cooldown=0,
143
                                    patience=5.
144
                                    verbose=1,
145
                                   min_lr=0.5e-6)
146
147 # Save weights
148 checkpoint = ModelCheckpoint(filepath=filepath,
```

```
149
                                  monitor='val loss'.
150
                                  verbose=1,
151
                                  save_best_only=True)
152
153 # Mean Square Error (MSE) loss function, Adam optimizer
154 autoencoder.compile(loss='mse', optimizer='adam')
155
156 # Called every epoch
157 callbacks = [lr_reducer, checkpoint]
158
159 # Train the autoencoder
160 autoencoder.fit(x_train_gray,
161
                    x_train,
                    validation_data=(x_test_gray, x_test),
162
163
                    epochs=30,
164
                    batch_size=batch_size,
165
                    callbacks=callbacks)
166
167\ {\rm \#\ Predict\ the\ autoencoder\ output\ from\ test\ data}
168 x_decoded = autoencoder.predict(x_test_gray)
169
170 # Display the 1st 100 colorized images
171 datasetGtoC = x_decoded[:100]
172 datasetGtoC = datasetGtoC.reshape((10, 10, img_rows, img_cols, channels))
173 datasetGtoC = np.vstack([np.hstack(i) for i in datasetGtoC])
174 plt.figure()
175 plt.axis('off')
176 plt.title('Predicted Colorized Images')
177~{\rm plt.imshow(datasetGtoC, interpolation='none')}
178 plt.show()
```

5.1.2 Three Convolutional Layer Implementation Without Learning Rate Modification

```
1 '''Colorization autoencoder
2
3 Used to train gray scale images of CIFAR-10 dataset to make them colorized
4 ,,,
5
6 from __future__ import absolute_import
7 from __future__ import division
8 from __future__ import print_function
10 from tensorflow.keras.layers import Dense, Input
11 from tensorflow.keras.layers import Conv2D, Flatten
12 from tensorflow.keras.layers import Reshape, Conv2DTranspose
13\ {\rm from}\ {\rm tensorflow.keras.models}\ {\rm import}\ {\rm Model}
14 from tensorflow.python.keras.callbacks import ReduceLROnPlateau
15 from tensorflow.python.keras.callbacks import ModelCheckpoint
16 from tensorflow.keras.datasets import cifar10
17 from tensorflow.keras.utils import plot_model
18\ {\rm from\ tensorflow.keras\ import\ backend\ as\ K}
19
20 import numpy as np
21 import matplotlib.pyplot as plt
22 \text{ import os}
23
24 def rgb2gray(rgb):
25
       """Convert from color image (RGB) to grayscale.
       .....
26
27
       return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
28
29
30 # load the CIFAR10 data
31 (x_train, _), (x_test, _) = cifar10.load_data()
32
33 # input image dimensions
34 # we assume data format "channels_last"
35 img_rows = x_train.shape[1]
36 img_cols = x_train.shape[2]
37 channels = x_train.shape[3]
38
```

```
39 # create saved_images folder
 40 imgs_dir = 'saved_images'
41 save_dir = os.path.join(os.getcwd(), imgs_dir)
42 if not os.path.isdir(save_dir):
43
            os.makedirs(save_dir)
44
45 # display the 1st 100 input images (color and gray)
46 \text{ imgs} = x_{\text{test}}[:100]
47 imgs = imgs.reshape((10, 10, img_rows, img_cols, channels))
48 imgs = np.vstack([np.hstack(i) for i in imgs])
 49 plt.figure()
50 plt.axis('off')
51 plt.title('Test color images (Ground Truth)')
 52 plt.imshow(imgs, interpolation='none')
53 plt.savefig('%s/test_color.png' % imgs_dir)
54 plt.show()
55
56\ \text{\ensuremath{\#}} convert color train and test images to gray
57 x_train_gray = rgb2gray(x_train)
58 x_test_gray = rgb2gray(x_test)
59
60 # display grayscale version of test images
61 imgs = x_test_gray[:100]
 62 imgs = imgs.reshape((10, 10, img_rows, img_cols))
63 imgs = np.vstack([np.hstack(i) for i in imgs])
64 plt.figure()
65 plt.axis('off')
66 plt.title('Test gray images (Input)')
67 plt.imshow(imgs, interpolation='none', cmap='gray')
68 plt.savefig('%s/test_gray.png' % imgs_dir)
69 plt.show()
70
71
72 # normalize output train and test color images
73 x_train = x_train.astype('float32') / 255
74 \text{ x_test} = \text{x_test.astype}('float32') / 255
75
 76 # normalize input train and test grayscale images
 77 x_train_gray = x_train_gray.astype('float32') / 255
 78 x_test_gray = x_test_gray.astype('float32') / 255
79
80 # reshape images to row x col x channel for CNN output/validation
81 x_train = x_train.reshape(x_train.shape[0], img_rows, img_cols, channels)
82 x_test = x_test.reshape(x_test.shape[0], img_rows, img_cols, channels)
83
84 # reshape images to row x col x channel for CNN input
85 x_train_gray = x_train_gray.reshape(x_train_gray.shape[0], img_rows, img_cols, 1)
86 x_test_gray = x_test_gray.reshape(x_test_gray.shape[0], img_rows, img_cols, 1)
87
88 # network parameters
89 input_shape = (img_rows, img_cols, 1)
90 \text{ batch_size} = 32
91 \text{ kernel_size} = 3
92 \text{ latent_dim} = 256
93 # encoder/decoder number of CNN layers and filters per layer
94 layer_filters = [64, 128, 256]
95
96 # build the autoencoder model
97 # first build the encoder model
98 inputs = Input(shape=input_shape, name='encoder_input')
99 x = inputs
100 # stack of Conv2D(64) - Conv2D(128) - Conv2D(256)
101 for filters in layer_filters:
102
       x = Conv2D(filters=filters,
103
                   kernel_size=kernel_size,
104
                   strides=2.
105
                   activation='relu',
106
                   padding='same')(x)
107
108 # shape info needed to build decoder model so we don't do hand computation
109 # the input to the decoder's first Conv2DTranspose will have this shape
```

```
110 # shape is (4, 4, 256) which is processed by the decoder back to (32, 32, 3)
111 shape = K.int_shape(x)
112
113 # generate a latent vector
114 x = Flatten()(x)
115 latent = Dense(latent_dim, name='latent_vector')(x)
116
117 # instantiate encoder model
118 encoder = Model(inputs, latent, name='encoder')
119 encoder.summary()
120
121 # build the decoder model
122 latent_inputs = Input(shape=(latent_dim,), name='decoder_input')
123 x = Dense(shape[1]*shape[2]*shape[3])(latent_inputs)
124 x = \text{Reshape}((\text{shape}[1], \text{shape}[2], \text{shape}[3]))(x)
125
126 # stack of Conv2DTranspose(256)-Conv2DTranspose(128)-Conv2DTranspose(64)
127 for filters in layer_filters[::-1]:
128
       x = Conv2DTranspose(filters=filters,
129
                             kernel_size=kernel_size,
130
                             strides=2,
131
                             activation='relu',
132
                             padding='same')(x)
133
134 outputs = Conv2DTranspose(filters=channels,
135
                               kernel_size=kernel_size,
136
                               activation='sigmoid',
137
                               padding='same',
138
                               name='decoder_output')(x)
139
140~\text{\#} instantiate decoder model
141 decoder = Model(latent_inputs, outputs, name='decoder')
142 decoder.summary()
143
144 # autoencoder = encoder + decoder
145 # instantiate autoencoder model
146 autoencoder = Model(inputs, decoder(encoder(inputs)), name='autoencoder')
147 autoencoder.summary()
148
149 # prepare model saving directory.
150 save_dir = os.path.join(os.getcwd(), 'saved_models')
151 model_name = 'colorized_ae_model.{epoch:03d}.h5'
152 if not os.path.isdir(save_dir):
153
           os.makedirs(save_dir)
154 filepath = os.path.join(save_dir, model_name)
155
156 # save weights for future use (e.g. reload parameters w/o training)
157 checkpoint = ModelCheckpoint(filepath=filepath,
                                  monitor='val_loss',
158
159
                                  verbose=1,
160
                                  save_best_only=True)
161
162 # Mean Square Error (MSE) loss function, Adam optimizer
163 autoencoder.compile(loss='mse', optimizer='adam')
164
165 # called every epoch
166 callbacks = [checkpoint]
167
168 # train the autoencoder
169 autoencoder.fit(x_train_gray,
170
                    x_train,
171
                    validation_data=(x_test_gray, x_test),
172
                    epochs=30,
173
                    batch_size=batch_size,
174
                    callbacks=callbacks)
175
176~\text{\#} predict the autoencoder output from test data
177 x_decoded = autoencoder.predict(x_test_gray)
178
179 # display the 1st 100 colorized images
180 \text{ imgs} = x\_\text{decoded}[:100]
```

```
181 imgs = imgs.reshape((10, 10, img_rows, img_cols, channels))
182 imgs = np.vstack([np.hstack(i) for i in imgs])
183 plt.figure()
184 plt.axis('off')
185 plt.title('Colorized test images (Predicted)')
186 plt.imshow(imgs, interpolation='none')
187 plt.savefig('%s/colorized.png' % imgs_dir)
188 plt.show()
```

5.1.3 Linear and Convolutional Latent Space Implementations

```
1 #!/usr/bin/env python
 2 \text{ # coding: utf-8}
 3
4 # In[1]:
 5
6
7 import time
8 import torch
9 import torch.nn as nn
10 \ {\rm import \ torch.nn.functional} as F
11 import matplotlib.pyplot as plt
12 \text{ import numpy as np}
13 import random
14\ {\rm from\ torchsummary\ import\ summary\ }
15
16
17 # In[2]:
18
19
20 # Information about device
21 print(torch.cuda.device_count())
22 print(torch.cuda.current_device())
23 \text{ print(torch.cuda.device(0))}
24 print(torch.cuda.get_device_name(0))
25
26 use_cuda = torch.cuda.is_available()
27 print(use_cuda)
28 # Set proper device based on cuda availability
29 device = torch.device("cuda" if use_cuda else "cpu")
30 \text{ print("Torch device selected: ", device)}
31
32
33 # In[3]:
34
35
36 # Construct Autoencoder
37 class Net1(nn.Module):
38
       def __init__(self):
39
           super(Net1, self).__init__()
40
41
           # Define layers of the autoencoder neural network
42
43
           # Encoder
           self.conv1 = nn.Conv2d(1, 64, 3, padding=1)
44
           self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
45
46
           self.maxpool1 = nn.MaxPool2d(2)
47
           self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
48
           self.maxpool2 = nn.MaxPool2d(2)
49
           # Code (Latent Representation)
50
51
           self.fc1 = nn.Linear(256*8*8, 256)
           self.fc2 = nn.Linear(256, 256*8*8)
52
53
54
           # Decoder
55
           self.upsample1 = nn.Upsample(scale_factor=2, mode='bilinear')
56
           self.deconv1 = nn.ConvTranspose2d(256, 128, 3, padding=1)
57
           self.upsample2 = nn.Upsample(scale_factor=2, mode='bilinear')
58
           self.deconv2 = nn.ConvTranspose2d(128, 64, 3, padding=1)
           self.deconv3 = nn.ConvTranspose2d(64, 3, 3, padding=1)
59
60
```

```
61
       def forward(self, X):
62
63
            # Encoder
            output = F.relu(self.conv1(X))
 64
65
            output = F.relu(self.maxpool1(self.conv2(output)))
66
            output = F.relu(self.maxpool2(self.conv3(output)))
67
68
            # Latent Representation
            output = torch.flatten(output,1) # Flatten
 69
 70
            output = F.relu(self.fc1(output))
 71
            output = F.relu(self.fc2(output))
72
            output = torch.reshape(output, (-1, 256, 8, 8)) # Reshape
73
 74
            # Decoder
 75
            output = F.relu(self.deconv1(self.upsample1(output)))
 76
            output = F.relu(self.deconv2(self.upsample2(output)))
77
            output = F.sigmoid(self.deconv3(output))
 78
79
            return output
80
81 # Construct Autoencoder
82 class Net2(nn.Module):
83
       def __init__(self):
84
            super(Net2, self).__init__()
85
86
            # Define layers of the autoencoder neural network
87
88
            # Encoder
89
            self.conv1 = nn.Conv2d(1, 64, 3, padding=1)
90
            self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
91
            self.maxpool1 = nn.MaxPool2d(2)
92
            self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
93
94
            # Code (Latent Representation)
95
            self.conv4 = nn.Conv2d(256, 256, 3, padding=1)
96
97
            # Decoder
            self.deconv1 = nn.ConvTranspose2d(256, 128, 3, padding=1)
98
99
            self.upsample1 = nn.Upsample(scale_factor=2, mode='bilinear')
100
            self.deconv2 = nn.ConvTranspose2d(128, 64, 3, padding=1)
101
            self.deconv3 = nn.ConvTranspose2d(64, 3, 3, padding=1)
102
103
        def forward(self, X):
104
105
            # Encoder
106
            output = F.relu(self.conv1(X))
107
            output = F.relu(self.maxpool1(self.conv2(output)))
108
            output = F.relu(self.conv3(output))
109
110
            # Latent Representation
111
            output = self.conv4(output)
112
113
            # Decoder
114
            output = F.relu(self.deconv1(output))
115
            output = F.relu(self.deconv2(self.upsample1(output)))
116
            output = F.sigmoid(self.deconv3(output))
117
118
            return output
119
120
121 # In[4]:
122
123
124 import torchvision
125 \ {\rm import\ torchvision.transforms\ as\ transforms}
126 import torch.optim as optim
127 from torchvision.transforms.functional import rgb_to_grayscale
128
129
130 # In[5]:
131
```

```
132
133 # Transforms
134 train_transform = transforms.Compose(
        [transforms.ToTensor(),
135
136
        transforms.RandomRotation(15)]
137
138
139 test_transform = transforms.Compose(
        [transforms.ToTensor()]
140
141
        )
142
143 # Load in CIFAR10 data, splitting training and test data and applying transforms
144 train_data = torchvision.datasets.CIFAR10("./", train= True, transform =
       train_transform, download = True)
145 test_data = torchvision.datasets.CIFAR10("./", train= False, transform =
       test_transform, download = True)
146
147 # Create dataloaders for loading in data in batches of size <code>mini_batch_size</code>
148 \text{ mini_batch_size} = 100
149 train_loader = torch.utils.data.DataLoader(train_data, batch_size=mini_batch_size,
       shuffle=True, num_workers=2)
150 test_loader = torch.utils.data.DataLoader(test_data, batch_size=mini_batch_size,
       shuffle=False, num_workers=2)
151
152
153 # In[6]:
154
155
156 # Define the training function
157 def train(model, num_epochs, optimizer, loss_func):
        # Set model to training mode
158
       model.train()
159
160
       losses = []
161
162
       for i in range(num_epochs):
163
            total_loss = 0
164
165
            for (images, _) in train_loader:
166
                images = images.to(device) # Push tensors to GPU
167
                grayscale_images = rgb_to_grayscale(images) # Convert images to grayscale
168
                optimizer.zero_grad() # Zero the gradients
169
                outputs = model(grayscale_images) # Forward pass
170
                loss = loss_func(outputs, images) # Calculate loss
171
                loss.backward() # Backpropagation
172
                optimizer.step() # Update weights
173
                total_loss += loss.item() # Accumulate loss
174
175
            # Save loss for graphing
176
            losses.append(total_loss / len(train_loader))
177
            print(f'(Epoch {i+1}) Training Loss: {total_loss / len(train_loader)} Test
       Loss: {test(model)}')
178
       return losses
179
180
181 # Define function for applying model to test data and returning the accuracy
182 def test(model):
183
        # Set model to evaluation mode
184
       model.eval()
185
       total loss = 0
186
187
        # Test on test set
        for (images, _) in test_loader:
188
189
            images = images.to(device) # Push tensors to GPU
190
            grayscale_images = rgb_to_grayscale(images) # Convert images to grayscale
191
            outputs = model(grayscale_images) # Forward pass
192
            loss = F.mse_loss(outputs, images) # Calculate loss
193
            total_loss += loss.item() # Accumulate loss
194
195
       return total_loss/len(test_loader)
196
197
```

```
198 # In[7]:
199
200
201 # Define hyperparameters
202 learning_rate = 1e-4
203 \text{ num\_epochs} = 20
204
205
206 # In[10]:
207
208
209~\text{\#} Create the model with FC latent code
210 net1 = Net1().to(device)
211
212~\text{\#} Print a summary of the model WIP
213 summary(net1, (1, 32, 32))
214
215~\mbox{\it \#} Define the loss function and optimizer
216 loss_func = nn.MSELoss()
217 optimizer = optim.Adam(net1.parameters(), lr=learning_rate)
218
219 # Load model, if saved parameters exist
220 try:
221
        net1.load_state_dict(torch.load("./saved models/net1.pth"))
222 except:
223
        # Perform training
224
        training_losses = train(net1, num_epochs, optimizer, loss_func)
225
226
        # Save model
227
       torch.save(net1.state_dict(), './saved models/net1.pth')
228
229\ \text{\#} Prepare model for evaluation
230 net1.eval()
231
232
233 # In[11]:
234
235
236 # Plot the losses over epochs for model 1 \,
237 plt.plot(training_losses)
238 plt.title("Training Loss vs Epochs")
239 plt.xlabel("# Epochs")
240 plt.ylabel("Loss")
241 plt.show()
242
243
244 # In[12]:
245
246
247~\text{\#} Create the model with CNN latent code
248 net2 = Net2().to(device)
249
250~\text{\#} Print a summary of the model WIP
251 summary(net2, (1, 32, 32))
252
253 \mbox{\it \#} Define the loss function and optimizer
254 loss_func = nn.MSELoss()
255 optimizer = optim.Adam(net2.parameters(), lr=learning_rate)
256
257 # Load model, if saved parameters exist
258\ {\rm try} :
259
       net2.load_state_dict(torch.load("./saved models/net2.pth"))
260 except:
261
        # Perform training
262
        training_losses = train(net2, num_epochs, optimizer, loss_func)
263
264
        # Save model
265
        torch.save(net2.state_dict(), './saved models/net2.pth')
266
267 # Prepare model for evaluation
268 \text{ net2.eval}()
```

```
269
270
271 # In[13]:
272
273
274 # Plot the losses over epochs for model 2
275 plt.plot(training_losses)
276 plt.title("Training Loss vs Epochs")
277 plt.xlabel("# Epochs")
278 plt.ylabel("Loss")
279 plt.show()
280
281
282 # In[]:
283
284
285 # Grab images for testing
286 test_images, _ = next(iter(test_loader))
287
288
289 # In[]:
290
291
292 ### Test Net1 ###
293
294 # Display images to be tested
295 display_test = np.array([transforms.ToPILImage()(img) for img in test_images])
296 display_test = display_test.reshape((10, 10, 32, 32, 3))
297 display_test = np.vstack([np.hstack(i) for i in display_test])
298 plt.figure()
299 plt.axis('off')
300 plt.title('Test Images (RGB)')
301 plt.imshow(display_test, interpolation='none')
302 plt.show()
303
304 # Convert RGB test images into grayscale
305 grayscale_images = rgb_to_grayscale(test_images).to(device)
306
307 # Display images in grayscale
308 display_test = np.array([transforms.ToPILImage()(img) for img in grayscale_images])
309 display_test = display_test.reshape((10, 10, 32, 32, 1))
310 display_test = np.vstack([np.hstack(i) for i in display_test])
311 plt.figure()
312 plt.axis('off')
313 plt.title('Test Images (Grayscale)')
314 plt.imshow(display_test, interpolation='none', cmap='gray')
315 \text{ plt.show()}
316
317 # Run model on grayscale images
318 output = net1(grayscale_images)
319
320 # Display images after decoding
321 display_test = np.array([transforms.ToPILImage()(img) for img in output])
322 display_test = display_test.reshape((10, 10, 32, 32, 3))
323 display_test = np.vstack([np.hstack(i) for i in display_test])
324 plt.figure()
325 plt.axis('off')
326 plt.title('Test Images (FC Code Autoencoder)')
327 plt.imshow(display_test, interpolation='none')
328 plt.show()
329
330
331 # In[]:
332
333
334 ### Test Net2 ###
335
336 # Display images to be tested
337 display_test = np.array([transforms.ToPILImage()(img) for img in test_images])
338 display_test = display_test.reshape((10, 10, 32, 32, 3))
339 display_test = np.vstack([np.hstack(i) for i in display_test])
```

```
340 plt.figure()
341 plt.axis('off')
342 plt.title('Test Images (RGB)')
343 plt.imshow(display_test, interpolation='none')
344 plt.show()
345
346 # Convert RGB test images into grayscale
347 grayscale_images = rgb_to_grayscale(test_images).to(device)
348
349 # Display images in grayscale
350 display_test = np.array([transforms.ToPILImage()(img) for img in grayscale_images])
351 display_test = display_test.reshape((10, 10, 32, 32, 1))
352 display_test = np.vstack([np.hstack(i) for i in display_test])
353 plt.figure()
354 plt.axis('off')
355 plt.title('Test Images (Grayscale)')
356 plt.imshow(display_test, interpolation='none', cmap='gray')
357 plt.show()
358
359~\text{\#} Run model on grayscale images
360 output = net2(grayscale_images)
361
362 # Display images after decoding
363 display_test = np.array([transforms.ToPILImage()(img) for img in output])
364 display_test = display_test.reshape((10, 10, 32, 32, 3))
365 display_test = np.vstack([np.hstack(i) for i in display_test])
366 plt.figure()
367 plt.axis('off')
368 plt.title('Test Images (Fully CNN Autoencoder)')
369 plt.imshow(display_test, interpolation='none')
370 plt.show()
```

5.2 My Contribution

In our pursuit of grayscale image colorization, my primary contribution revolved around implementing a pivotal component: the 3-layer convolutional autoencoder. Our team collectively determined autoencoders as the optimal approach for this task, and I undertook the responsibility of translating this decision into a functional model using TensorFlow and Keras. Despite my initial unfamiliarity with these tools, extensive research and learning enabled the successful implementation of the model. A notable aspect of my contribution was incorporating a fixed learning rate into the model architecture, aiming to discern its impact on image quality. Beyond implementation, I played a pivotal role in shaping crucial sections of the report, specifically contributing significantly to the Results, Analysis, Conclusion, and Contribution segments, section 5.1.2. This encompassed synthesizing and interpreting outcomes, drawing meaningful insights from the experiments conducted, and offering valuable input into the implications and future directions derived from our findings.