COS 429 Final Project
Final Report

Utilizing Cross-Channel Autoencoders for Deep Learning Image Colorization

Team Members:
1. Tyler Kaye (tkaye)
2. Harry Heffernan (hmlh)
3. Mitchell Hamburger (mh20)

Abstract:

In this paper we detail the building and training of a cross-channel autoencoder capable of coloring black and white photographs. An important distinction is that our network is optimized not to recover the ground-truth colorization of the image, but rather to output a *plausible* colorization of the image. Many previous studies have attempted to find the ‘best’ or ‘averaged’ output of the network, but often this leads to heavily de-saturated images. This de-saturation is a result of using specific loss functions that encourage conservative color predictions in order to minimize color variance. Thus, we will compare and contrast three fully trained models each with a unique loss function specifically tailored to output brighter and thus more realistic colors. Several of the other key challenges included selecting a color-space representation, pre-processing the images, designing a network architecture, and evaluating our results. Finally, rather than training a complex network on a very large collection of images of various different categories, we instead focussed on creating a smaller network that trains only on a subset of images for which we are trying to colorize (landscapes, portraits, animals, etc). This optimized the results of our network as well as decreased the computational complexity of training and testing the network. If we are trying to colorize old landscape photos, there is no reason that we should be feeding it unrelated and extraneous photos as training inputs. Instead of training a large network to poorly color every possible image, we have created a framework to train a highly specialized network for a specific category of images. Ultimately we were able to create a very successful image colorizer capable of training on a dataset of several thousand images in under three hours.

Motivation:

This project enables us to not just take color photos and re-colorize them, but more importantly it allows us to add color to older photographs taken with cameras that were unable to capture color. In this way, we can view historical moments and even our own older photographs in fully realistic color. We hope that this will allow younger people to imagine how the landscapes and people looked in the past, and provide older generations with a sense of nostalgia as we present images that more closely resemble their own memories.

Previous Work:

Several researchers have approached the topic of image colorization using deep learning. Following the creation of the reddit page “Colorization” in 2014 in which people used photoshop to color images, many researchers have tried to create more generalized solutions to the problem of coloring old images using large datasets of colored photos. *Dahl* was one of the first researchers to approach the problem by using a pre-trained model that won the ILSVRC challenge. This model was used to extract a set of hyper-columns in order to begin learning the colorization from the output of the classification network. Ultimately Dahl used an unsatisfactory loss function which led to some disappointing results such as the sepia-toned building and sky in Figure 1. Furthermore, as the model minimizes the overall loss, instead of choosing a color for something like a car that can be many colors, as seen in Figure 2 it just averages the possible colors into a brownish-grey.

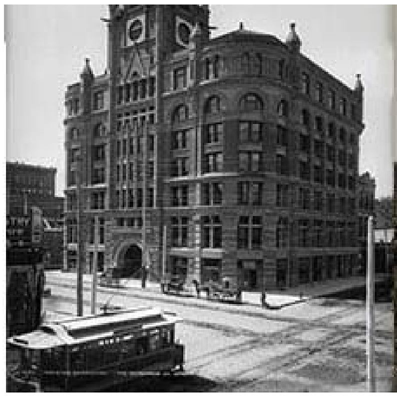


Figure 1.1: Input

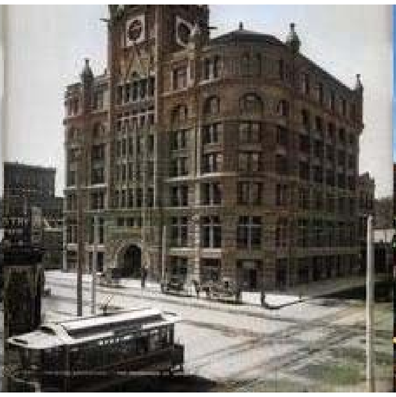


Figure 1.2: Output

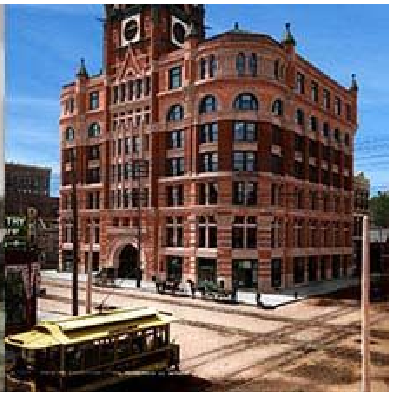


Figure 1.3: Ground Truth

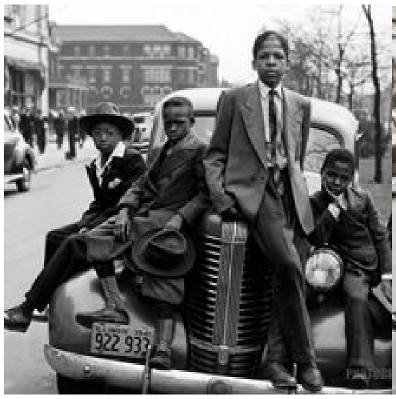


Figure 2.1: Input

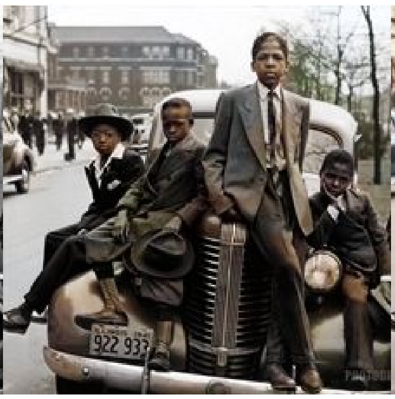


Figure 2.2: Output

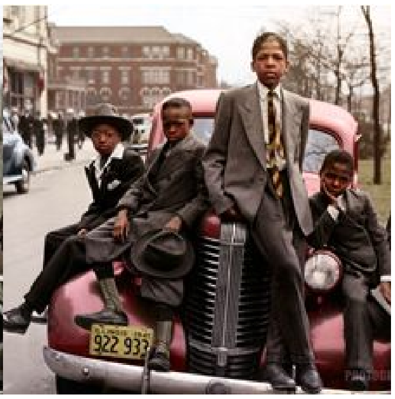


Figure 2.3: Ground Truth

Hwang et al. improves on this research by introducing a more sophisticated loss function, residual skip connections, and the use of the CIE LUV color space representation. This research utilized the Huber loss function and then maps the outputs to a discrete set of bins and outputs the bin with the highest probability of being correct. While this loss function was a step in the right direction, this model was unnecessarily large and in doing so the network could not be trained on a medium-sized dataset. As we are more interested in creating smaller networks that are highly specialized, this approach is ultimately unsatisfactory. Figure 3 shows the output of this network, and how its output (image in the middle) often was very muted in color. Thus, although their method of binning the individual losses before assigning outputs was a step in the right direction, it still led the model to consistently produce conservative color predictions leading to a bland image. Furthermore, their model often led to ‘color inconsistency’ problems, shown in Figure 4, in which it would color solid objects with different colors, such as a shirt as half one color and half another. This was an important consideration for our research, as we needed to further improve the loss function in order to avoid this problem. Lastly, the use of the CIE-LUV color-scheme was ultimately unsatisfactory for the researchers.

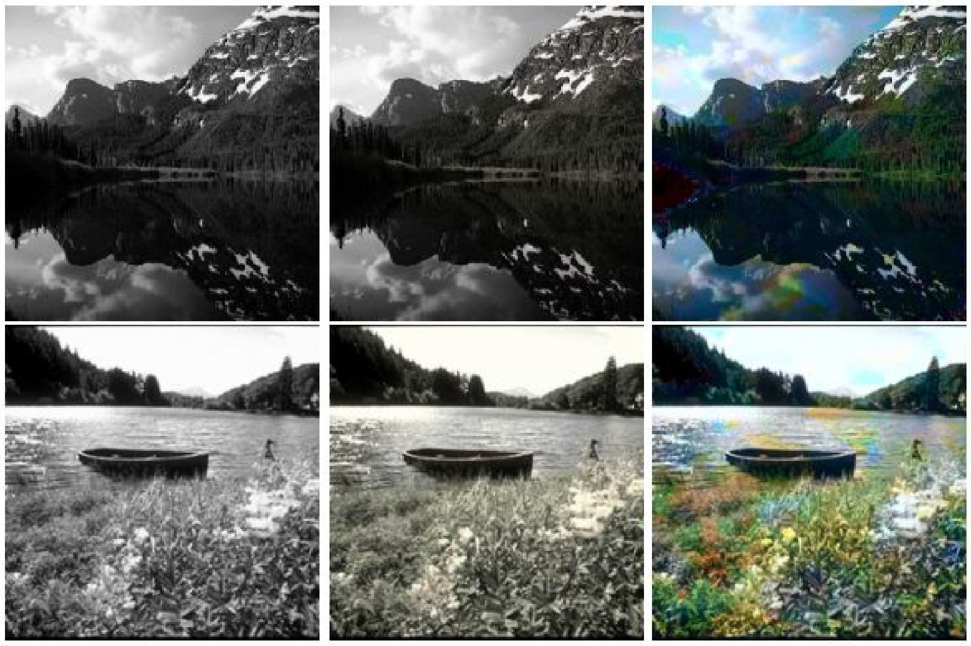


Figure 3: Input Output Ground Truth

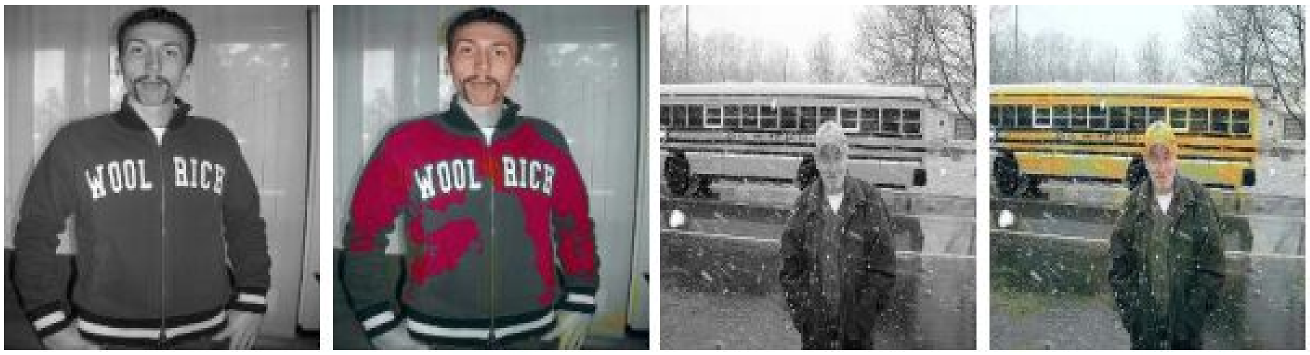


Figure 4: Color inconsistency Input Output

Several other researchers, including *Mao et al.* and *Deshpande et al.*, improved on this method, while keeping the main aspects of the model the same, by using pre-trained models as the first layers of their network. In addition, they had success while experimenting with different network architectures as well as loss functions. *Zhang et al.* was the first research to drastically diverge from much of the other research in this area. The researchers main aims were to solve the problem of an apple. As they describe it:

> “An apple is typically red, green, or yellow, but unlikely to be blue or orange. To approximately model the multimodal nature of the problem , we predict a distribution of possible color for each pixel. Furthermore, we re-weight the loss at training time to emphasize rare colors.” *(Zhang et al, 2)*

This research was highly focussed on the problem of creating more vibrant colors using a more sophisticated bucketing system and taking the annealed mean of the distribution outputted by the model. While this yielded slightly more promising results, it involved creating an entire network from scratch in order to learn a probability distribution for the coloring of an image. Although this did produce more vibrant colors, the network was entirely too large for the purposes of this project and it had some unsatisfactory results such as color inconsistency.

Selecting a Dataset:

In selecting a dataset for the training of our network, we sought a set of richly-colored images that are representative of the types of images people may want to convert from black-and-white into color. Many of the larger computer vision datasets such as ImageNet, PASCAL, and Caltech-256 feature objects with little to no background behind them. They serve their purpose for classification, but they do not provide us with the richly-colored images that we hope to recreate. Secondly, these images are not representative of the old images people would want to colorize using this project. Generally, older photographs are of people, cities, and landscapes; therefore, a dataset of images of thousands of different animals, kitchen appliances, and hundreds of different types of fungi will not be ideal for the purposes of this application. Therefore, we used the MIT CVCL Urban and Natural Scene dataset which contains images from eight categories: beach, country, forest, mountain, highway, street, city, and buildings. We chose to focus on landscapes because there is an abundance of old black-and-white landscape photographs and thus it would be a fitting set of images to train our specialized model on. However, the main idea underlying this decision is that we would rather train a smaller network to accurately learn to color a specific subset of images as opposed to employing a net that learns to color every possible image poorly.

Color-Scheme Representation:

There are many different color-scheme representations including RBG, Tristimulus, XYZ, HSV, CMYK, LAB, and NCS color models. We ultimately chose the LAB color space because the L channel represents the black-and-white projection of the image as it is the intensity, and then the AB channels add color to the image-representation. Thus, we can send in the L-channel as input to the network and have our model output its prediction for the A and B channels. Then these predictions can be compared to the ground-truth A and B channels of the image in order to backpropagate through the network. This separation is important as it enables us to perfectly model black-and-white images using the L channel and then simply overlay the A and B channels as a coloring on top of the L channel. Therefore, the main elements of the image (edges, corners, and blobs) remained where they should.

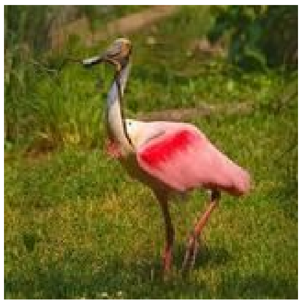


Figure 5.1

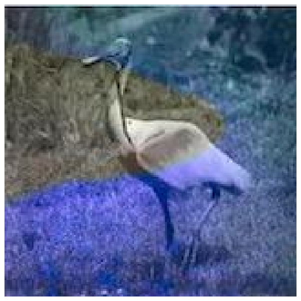


Figure 5.2

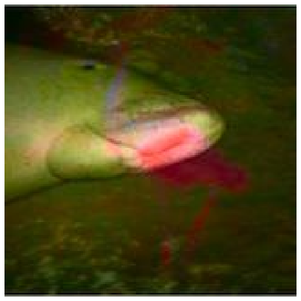


Figure 5.3

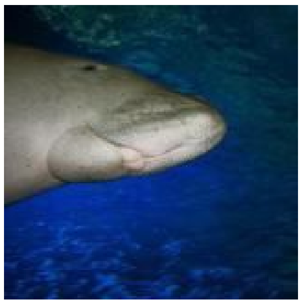


Figure 5.4

Figures 5.1 and 5.4 show two images from the training set with radically different colors. Figures 5.2 and 5.3 show the decomposition of the two images into the L and AB channels and then reconstructing the L channel of the first image with the AB channels of the second image and vice versa. As we can see, the AB channels just add coloring to the image and leave the main structure of the images in tact. This property is incredibly important because we do not want to have to reconstruct the details of the image (a significantly harder problem), but rather we just want to reconstruct the coloring of the images. The deconstruction of the L and AB channels can be seen below.

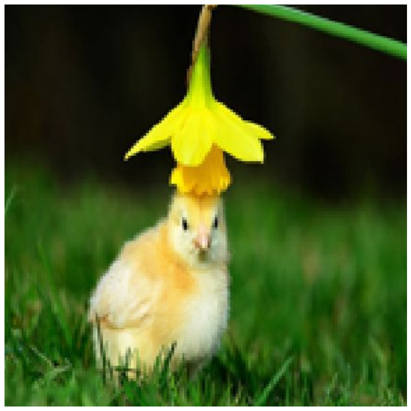


Figure 6.1: Original

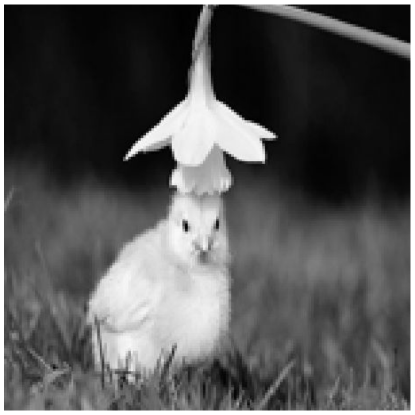


Figure 6.2: L Channel

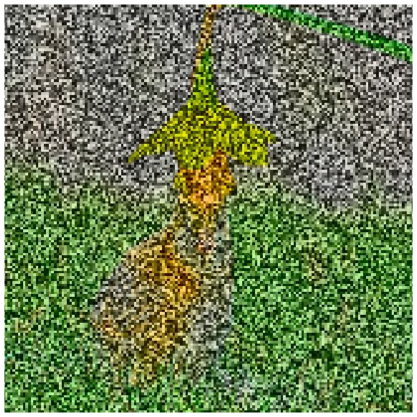


Figure 6.3: AB Channels

Above we see the original image in figure 8.1 followed by the just the image's L channel in figure 6.2. FInally in figure 6.3 we see the A and B channels on top of a randomized L channel. The above figures show that the decomposition of the LAB color-space is ideal for the purposes of our project as it perfectly separates the color of an image from its underlying blueprint.

Image Pre-Processing:

At first we began by subtracting the mean from the image (as is often done), but we ultimately achieved better results by simply reading in the image, converting it into the LAB color-scheme and then directly feeding the L channel into the network. Additionally, as the dataset does not come in a standardized size, we wrote a python script that converts all of the images to a specified size because we wanted to reduce the computational complexity of our network by decreasing the input size. As the dataset is too large to hold in memory, we had to continuously load the picture from its name. Therefore, we did the image resizing independently of the batch processing in order to further ease the computational demands of the training process.

That being said, it is important to point out that because our network is entirely composed of convolutional, pooling, and non-linearity layers, we can send in images of arbitrary size. The output will have the same dimensions as the inputs because for each of the pooling operations in the encoder section there is an matching transposed pooling operation that nullifies its effect on the image size. Even though we were afforded this flexibility, we choose to resize the images for training so that we could reduce the computational complexity of training. By making the images smaller and the same dimensions, the training could be optimized and thus the training time decreased. Furthermore, by making the images all the same size, we ensured that the model learned features at a specific scale which increased its accuracy.

Machine Learning Framework:

We initially began the development of this application using the MatConvNet framework built for MatLab; however, we soon found that this framework is designed for classification networks and the source code of the framework had to be edited in order to develop a cross-channel autoencoder. Although there was a recent addition to the framework that included a helpful pdist() function for autoencoders, the current implementation was unhelpful for our purposes. As it concatenated the channels of the mean-squared-error before propagating them backwards, its source code had to be drastically altered to support a cross-channel / multiple-channel autoencoder. Furthermore, the challenges associated with running MatLab in the cloud presented a problem when it came time to train the network. Therefore, the final application was developed using Tensorflow in order to take advantage of the ability to craft custom loss functions as well as the convenience of developing in a Jupyter Notebook.

<u>Machine Learning Technique:</u>

For this project, we utilized a cross-channel autoencoder to learn how to colorize the black-and-white images as inputs. Cross-channel autoencoders are a special class of autoencoders and thus neural networks. Neural networks tether layers of ‘nodes’ together by weighting different branches before applying a non-linearity function, often to classify the input. Convolutional neural networks use convolution, a mathematical operation in order to ease the computational burden of the network by paying more attention to where the input is in an image and and apply a kernel only to the pixels surrounding it. In this way, each neuron is not connected to all other neurons in the surrounding layers but rather only so a small subset of the neurons near it. Often, neural networks are used in supervised-learning in which the network’s output is compared to a ground truth output. Thus, an output (or rather a classification) is necessary for training a neural network.

However, autoencoders enable the network to perform unsupervised learning by using the input to the network as its own desired output. This is most often utilized as a form of dimensionality reduction by encoding the input information into a much smaller vector-space before decoding it back to its original input. However, this form of dimensionality reduction was the opposite of what we wanted our network to do. Rather than using a funnel-shape, we wanted to expand the input to more information, not less. Thus, our network can most aptly be described as a convolutional sparse autoencoder.

Lastly, this notion of feeding the L channel of the image into the network in order for it to learn the A and B channels of the inputted images is known as cross-channel autoencoding. We will be using one channel of the image in order to learn the remaining channels (in this case the color).

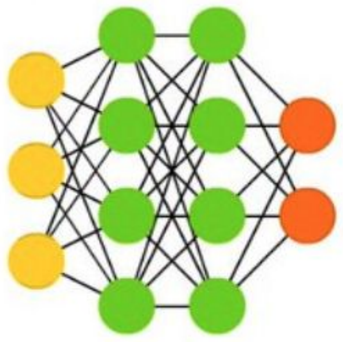


Fig 7.1: Deep Feedforward Network

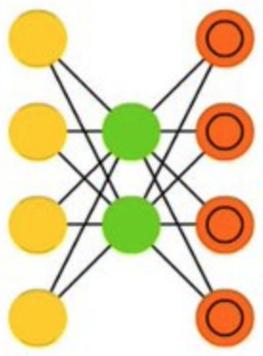


Fig 7.2: Autoencoder

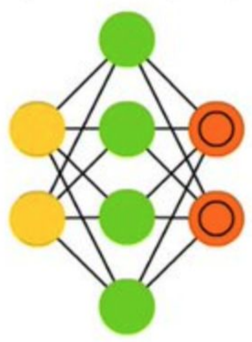


Fig 7.3: Sparse Autoencoder

Network Architecture:

The basic architecture of our network is summarized below in Figure 8. Initially we sought to develop a network capable of training on a moderately-sized dataset within a few hours; however, we found that training a complete network took way too long for the purposes of our application. Therefore, in order to quickly train a highly-specialized network we utilized the first 3 layers of the VGG-19 pre-trained model; each of which includes several rounds of convolutional filters, ReLu activations, and max-pooling operations. We inserted our own convolutional layer before the pre-trained model in order to send the required 64-channels into the network. Thus the beginning of the pre-trained model acted as our encoder. Then, we inserted our own custom layers of 4x4 kernel convolutional layers in order to expand the layers back to the proper size and output 2 channels representing the A and B channels. Finally, our network combined the 2 channels of the output with the L-channel of the input resulting in the fully-colored output image. This image was then compared to the ground-truth output and the loss function between the two images was minimized throughout training.

By using a pre-trained network as a part of our own network, we reduced training time as there are only 4 layers that our network had to learn throughout the training process. Training the encoder section of a network is incredibly challenging and time consuming as it learns a complex set of feature maps. In addition to aiding our training time, using a network trained to classify subsets of images was helpful as the pre-trained encoder was capable of identifying features such as trees, buildings, cars, mountains, etc. Thus our decoder was able to take this information and then learn to associate specific features with color-schemes.

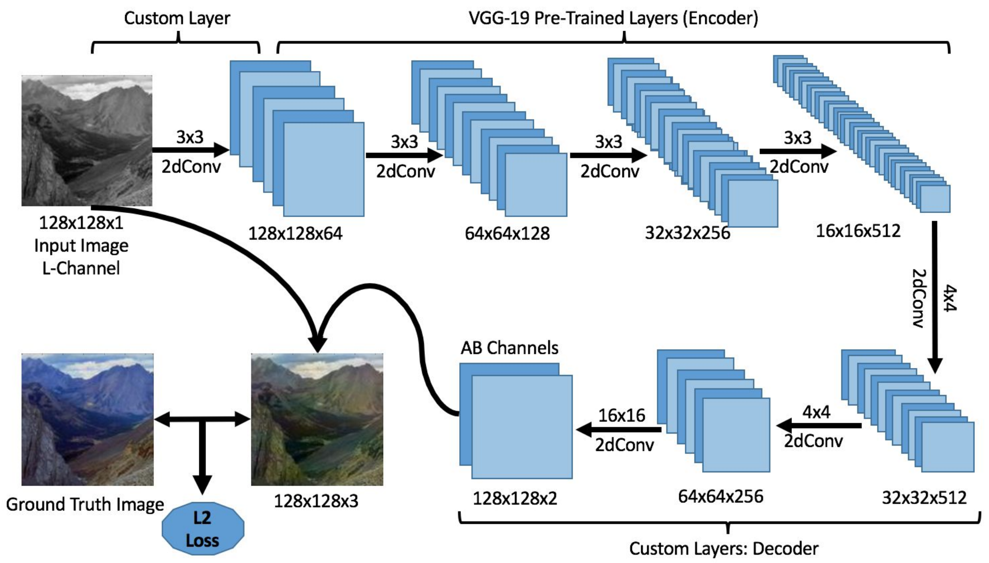


Figure 8: L2-Loss Network Architecture

As pointed out earlier, although the net is optimized for images of size 128x128, its use of strictly convolution, pooling, and ReLu layers enables it to work for images of arbitrary size.

Architecture Experimentation:

We experimented with the architecture quite a bit before coming to the above architecture. At the core of our decision was the tradeoff between size and accuracy. While we wanted to create an accurate and realistic colorizer, we also wanted to limit the size of the model in order to minimize training time. Our motivation was to create a smaller and more-specialized colorizer that was able to train on a few thousand images in a short time. However, as the model became bigger, not only did the training time increase, but also the number of images necessary for the model to learn the colorizations increased. Therefore, our experimentation led us to minimize the number of layers in our model to 4. The first layer was necessary in order to connect the inputs to the required 64 channels that the VGG model accepted. Next, we determined that a decoder consisting of 3 layers was the smallest that we could make it as we had to condense 512 channels into just 2.

Once we determined the number of layers, we then experimented with different kernel sizes. We came to use 4x4 kernels as the normal kernel size is anywhere between 3 and 5 pixels. Therefore, we selected 4 because we found it to be a good speed / accuracy tradeoff. Finally, we determined the number of channels per layer. We experimented with different values for the number of channels in the decoder layers, and ultimately came to 512 → 256 → 2. We initially used 256 → 128 → 2 but we found that the colorizations of those images were inferior to the configuration with more channels. While our configuration was more computationally taxing, we had already minimized so many other factors in our architecture that we felt this was a good tradeoff.

Training and Results:

We ultimately trained our network using three distinct loss functions: L2 Loss, Huber Loss, and Pairwise Mean Squared Error. The error functions and their training graphs are shown below in Figures 9, 10, and 11. The training time for all three loss functions took roughly 3 hours and we ran the training for 140 epochs as the progress began to flatten out. Furthermore, we ensured that the validation error did not diverge from the training error which would suggest that the model was overfitting.

L2 Loss Function:

$$L2_{loss} = \frac{1}{n_{pixels}} \sum_{i=0}^{width} \sum_{j=0}^{height} (Y_{ij} - \widehat{Y_{ij}})^2$$

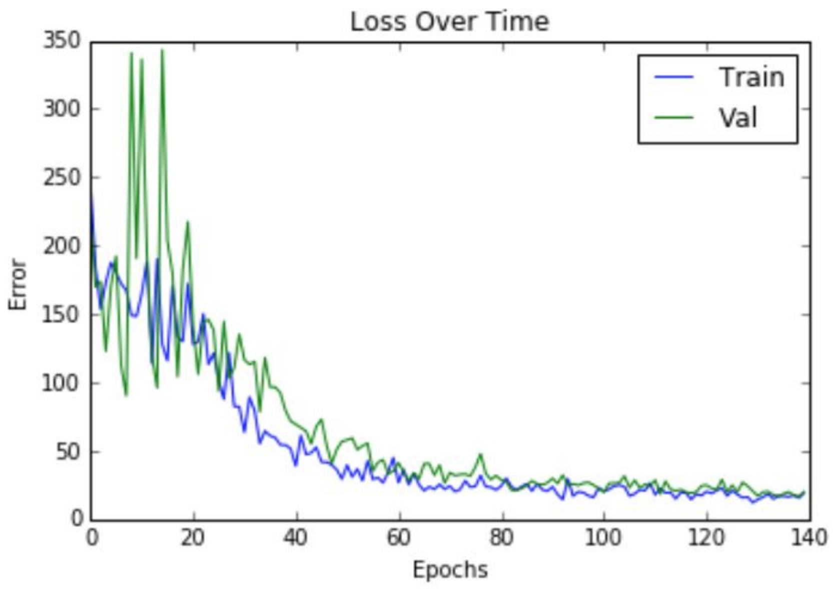


Figure 11: Training for L2 Loss

The loss can best be described as the average distance per-pixel, per-channel between the predicted values and the actual values. This suggests that towards the end of training we averaged a squared distance of roughly 25. This means that on average our estimated values for the A and B channels of each image were off by 5. As the A and B channel's values range from -128 to +127 this is very satisfactory as we can expect to find some color variation as discussed below.

Huber Loss:

$$Loss_{Huber} = \begin{cases} \frac{1}{2}\,(Labels - Predictions)^2 & if\;\; Labels - Predictions \leq \;\delta \\ \frac{1}{2}\delta^2 + \;\delta * (|Labels - Predictions| - \;\delta) & Otherwise \end{cases}$$

The Huber loss function is defined as a piecewise function such that if the residuals (Labels - Predictions) are less than some threshold δ then the loss function follows the L2 Loss function, but if the residual is larger than this threshold then it becomes the L1 loss function. By using this threshold δ we are able to extract the most important advantages from each individual loss function by being more robust to outliers (L1) while it de-emphasizing points the model has already fit closely to (L2).

Pairwise Mean Squared Error:

Unlike the Mean Squared Error loss function, which is a measure of the differences between corresponding elements of predictions and labels, Mean Pairwise Square Error is a measure of the differences between pairs of corresponding elements of predictions and labels. This loss function may be ideal for our purposes as it gives the individual losses a sense of location within the image.

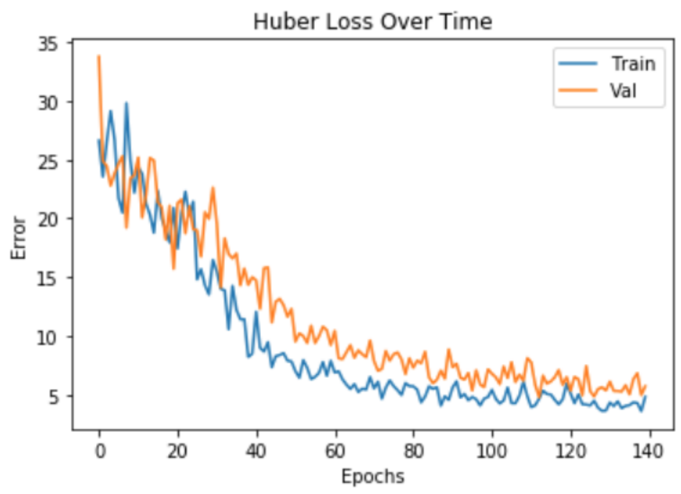


Figure 12: Training for Huber Loss

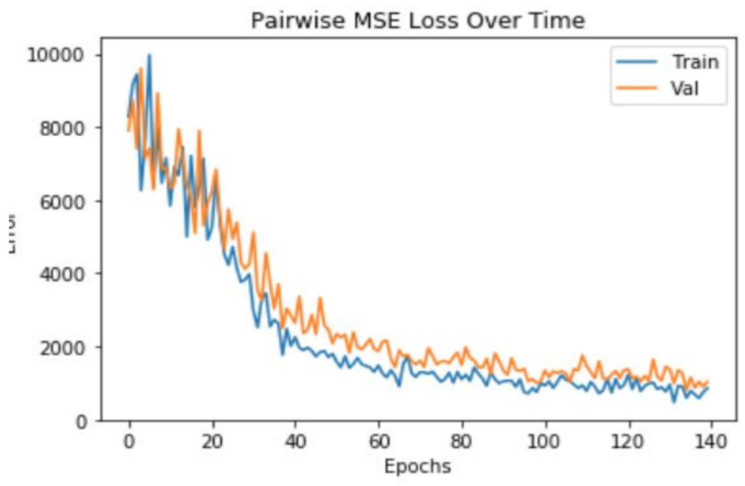
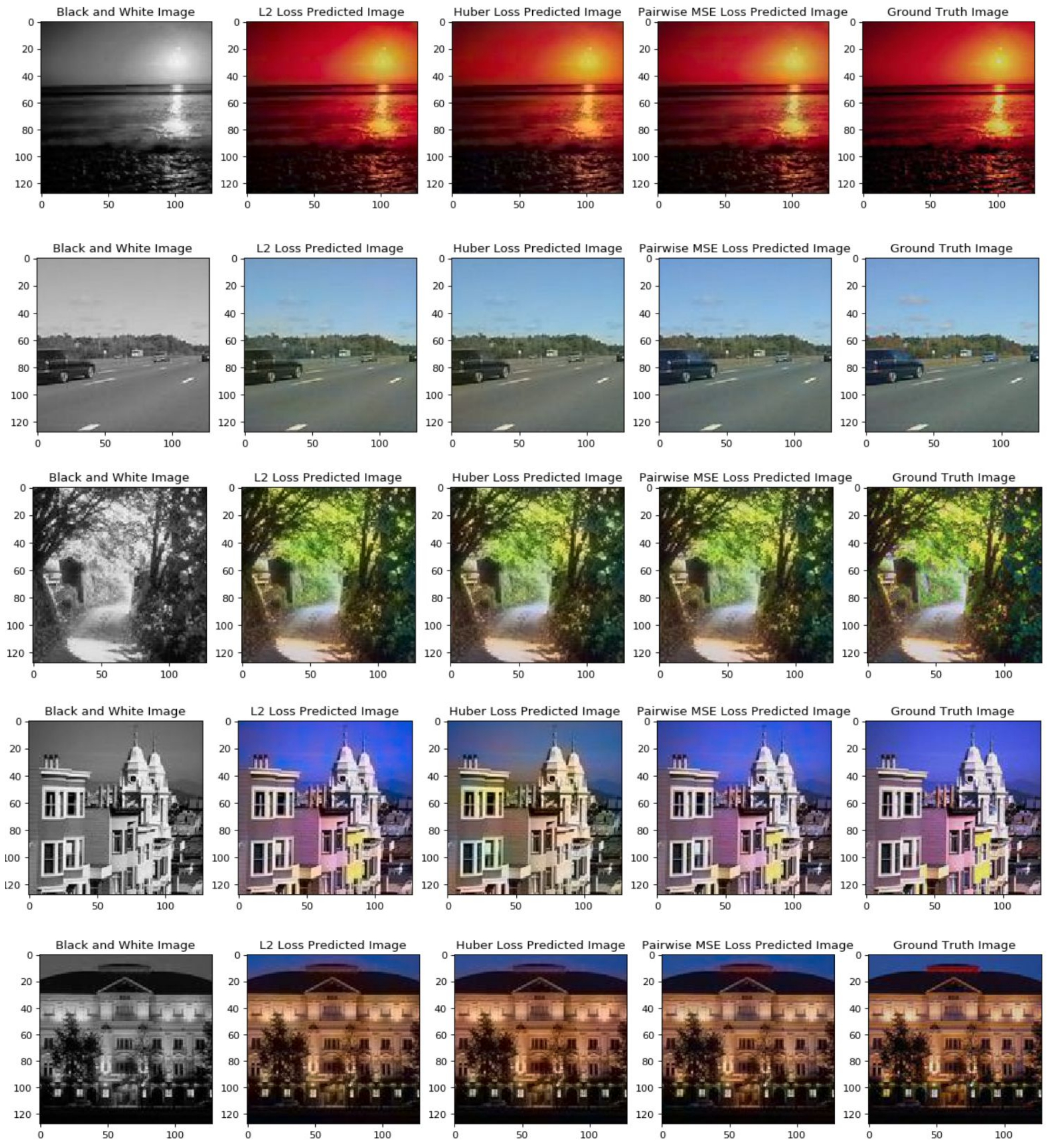


Figure 13: Training for Pairwise MSE

Note: As each of the above graphs represent the training loss of entirely different loss functions, their scales should not and cannot be compared.

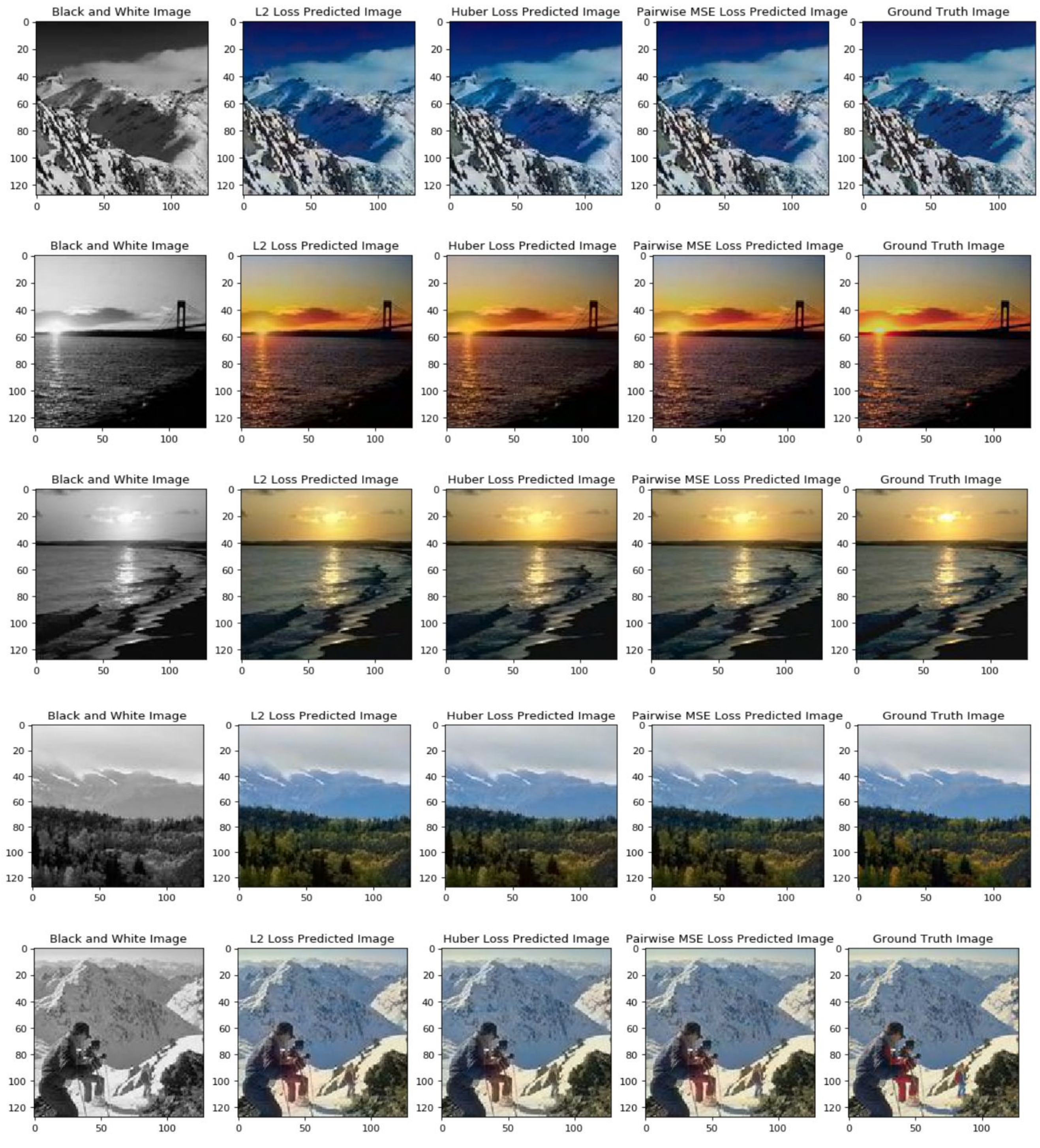
Training Set Results:



Input (B&W) Image - L2-Loss - Huber Loss - Pairwise MSE - Ground Truth

Figure 14: Training Set Examples

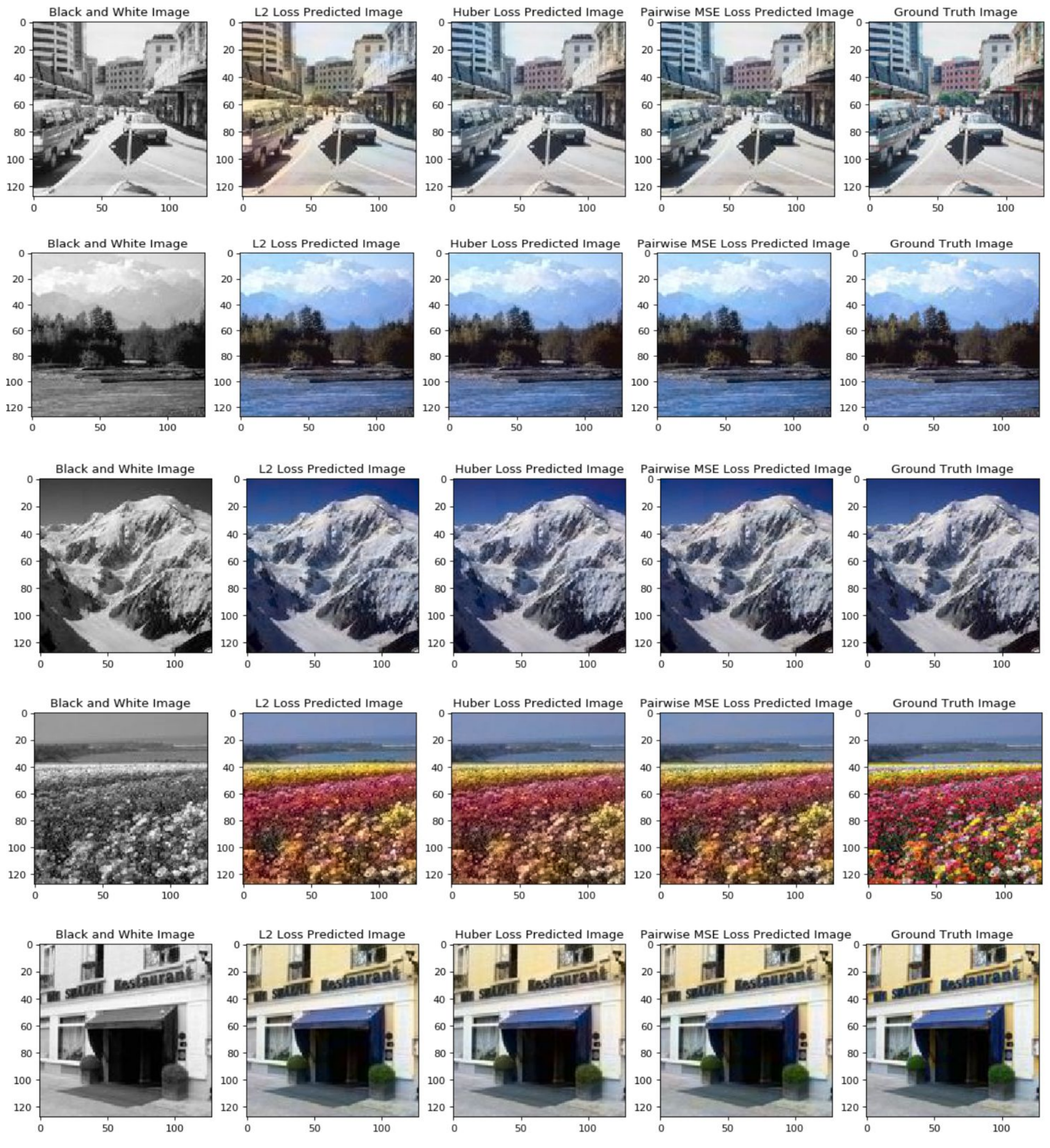
Validation Set Results:



Input (B&W) Image - L2-Loss - Huber Loss - Pairwise MSE - Ground Truth

Figure 15: Validation Set Examples

Testing Set Examples:



Input (B&W) Image - L2-Loss - Huber Loss - Pairwise MSE - Ground Truth

Figure 16: Testing Set Examples

Discussion:

A few patterns emerge in viewing the above results from the Training, Validation, and Testing sets. The first observation is that the models for all three loss functions seemed to really perform well on images of the sky, mountains, water, and trees. As these are heavily featured in the dataset, the model completely understands how and where to color them in the images. In fact, there are several images in the dataset that include trees in the fall with their burnt orange and reds, but the model still colors them as green. Taking into account that we are simply trying to output plausible images this seems very reasonable. There are very few distinguishing marks that differentiate fall foliage from the green spring / summer scenes in the black and white version of an image. Thus, because the fall images are underrepresented in the dataset, the model outputs what it knows to be a possible understanding of the image. As we are interested in a plausible colorization and not necessarily the right colorization this seems to be a perfectly sound output for the model.However, it is interesting to note that the model is able to identify and properly color varying intensities of sunsets in an image from the composition of the sky in black-and-white.

For most of the pictures of just scenery, all of the models appear to color the images quite well and there is not much variation in their behaviour; however, in images of the city (with secondary objects) the differences and weaknesses of the loss functions are revealed. The models often struggle to color secondary objects (clothing, cars, etc) in vibrant colors because it is trying to minimize the error of a non-probabilistic function. In so doing, the model is always always going to output a pseudo-conservative color value for any feature it has identified as taking on numerous colorings. *Zhang et al* attempted to solve this problem by having the model learn and minimize the error of a probability distribution. In so doing their model was able to choose maximums on the probability distribution that deterministic loss functions would not be able to output. However, as we are more focussed on creating a smaller model that trains on landscape images (in which secondary objects are sparse and often muted in color) this shortcoming was acceptable.

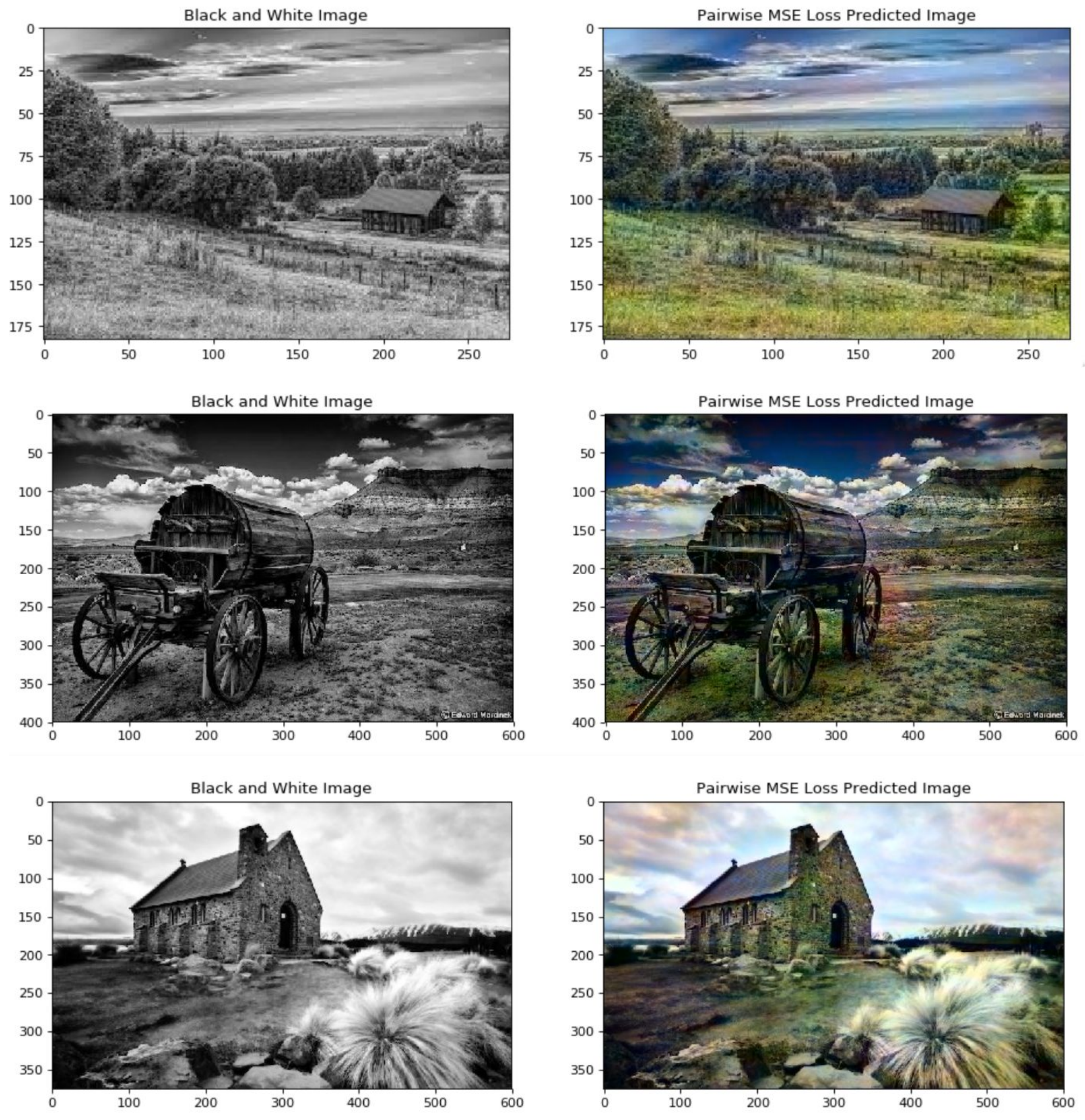
It is important to note though, that this limitation applied mostly to singular objects, and for things such as a patch of flowers the model was able to output a spectrum of different, vibrant colors. We see this in the fourth testing image in which the flowers take on rich colors and are not necessarily the same colors as the corresponding flowers in the ground-truth image.

Furthermore, although each of the loss functions struggled to color secondary objects, it does appear that the pairwise mean squared error was able to overcome this shortcoming at times by associating with the pixels around it to give objects slightly more vibrant colors. In the fifth photo of the validation set, we see that the output of the model for the pairwise mean squared error loss function was able to slightly color the people in the image, or at least more so than the other models. In addition we see this superiority in the second, fourth, and fifth images of the training set examples in which the car, buildings, and the roof, respectively, have a more vibrant color than in the prediction of the other loss functions. As the loss functions output values that cannot be compared because they represent the outputs of completely different functions, we compared and contrasted the success of the models utilizing different loss functions by visibly observing the colorizations by different models of the same input.

The pairwise mean squared error does not minimize the error of a pixel in the predicted output with its corresponding values in the ground-truth image but rather it minimizes the differences between the patch of pixels around it and the corresponding patch of the true colorization. Thus, it appears that minimizing this function was able to give the loss function a sense of its location and surrounding in the image that the other functions could not understand. This ability led to a visible advantage over the other loss functions in its ability to output more vibrant colorizations for secondary objects in the images. Due to this superiority over the other loss functions, for the final model, we selected to train using the pairwise mean squared error loss function.

True Only Black-And-White Images:

Below in Figure 17 are five original black-and-white photographs that have been run through our model in order to color them. In an attempt to not alter the images, they were not resized and thus their proportions remained intact as evidences by the labels on the images.


Black and White Image
Pairwise MSE Loss Predicted Image
Black and White Image
Pairwise MSE Loss Predicted Image
Black and White Image
Pairwise MSE Loss Predicted Image

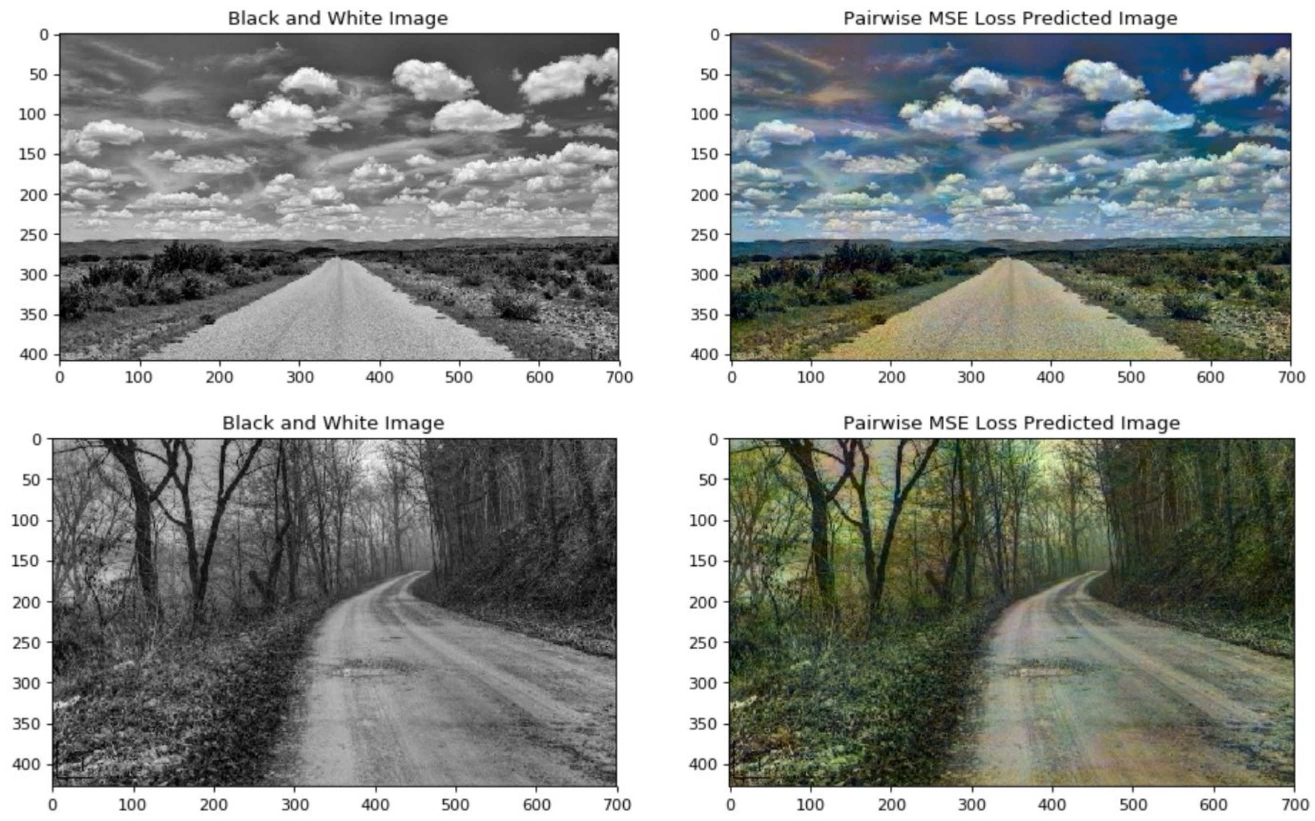

Black and White Image
Pairwise MSE Loss Predicted Image
Black and White Image
Pairwise MSE Loss Predicted Image

Figure 17: Transforming Originally Black and White Images

The above examples cannot be compared or contrasted to a ground truth because they do not exist. However, looking at the above examples, each one appears very realistic and they each contain vibrant colors in the plants, skies, and flowers. Out of curiosity, we ran a black and white cartoon through our model to observe if it would still work. The results can be viewed in Figure 18.

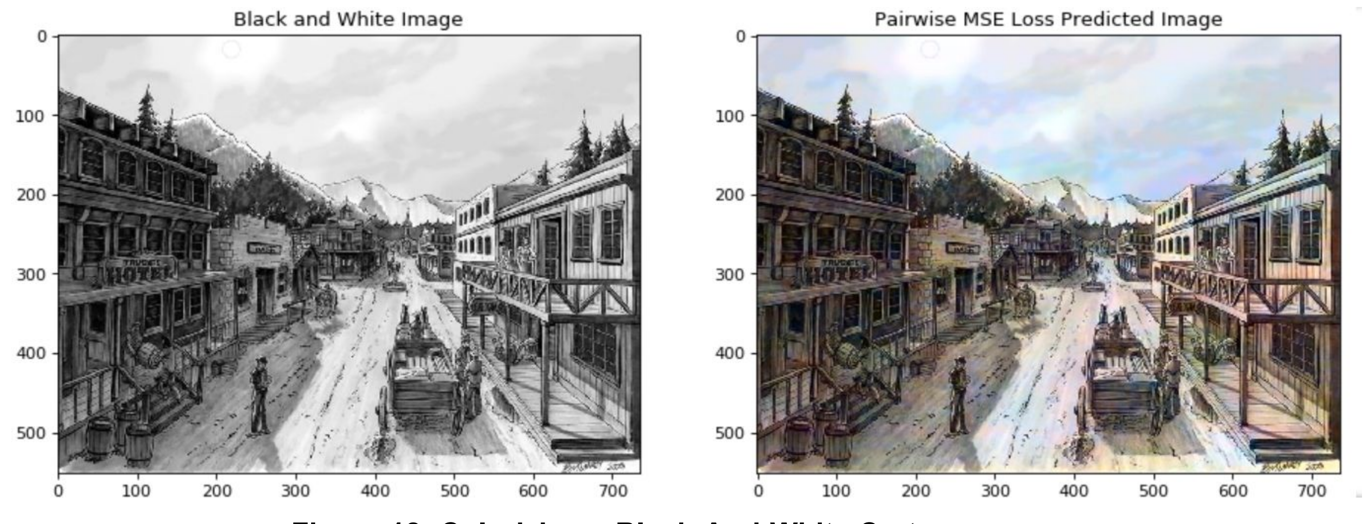


Figure 18: Colorizing a Black-And-White Cartoon

Conclusions:

Ultimately, we were very pleased with the results of this project. While others had previously completed work in this space, we were able to identify a unique approach that, in most cases, yielded superior results to those other projects. By focusing on one subset of photos, our cross-channel autoencoder was able to perform favorably on a specialized group of images. Specifically, the consistent coloration styles of landscape photographs made this a good choice for our training data. In addition, we were able to avoid some of the common pitfalls of previous work in the field. Our colorizations included vibrant colors, contrasting sharply with the muted sepia-toned efforts from previous researchers. We were also satisfied with the accuracy of our colors as compared to the related projects, as our model very accurately determined the correct shade of most objects. Despite these successes, however, our model did fail in its ability to evenly and thoroughly color objects. Though the ill-effects of this were far less drastic than examples such as those in Figure 4, our model did, to a degree, suffer from color inconsistency.

With the positive results from this project, it is certainly an endeavor that we want to continue with in the future. On a technical level, we would like to further experiment with our model and see if we can improve on it in any way. We could continue to customize loss functions in order to improve our results, possibly encountering one that functions better than pairwise MSE. We could also try to create a custom encoder, replacing the VVG-19 pretrained encoder that was used in this project. We also want to extend the work to categories other than landscapes. We think that the research could perform similarly well on other specific datasets. In particular, we would like to focus on other categories for which there is an abundance of old photographs. One area that may be interesting is still-lifes. This represents a large category of images, and while they are not as rich in color or in background as landscapes, we think that their relatively limited subset of objects would allow for very good performance.

Lastly, we see this as a potential iPhone application as we believe people would really enjoy utilizing this feature on social media sites.

Submitted Code:

Below we have highlighted the sections of code included with this report in order to further understand our research as well as use our trained model to colorize images.

ImageResizingScript/resize.py:

This is a small python script that resizes all of the images in a directory to a variable size defined in the program.

Colorizer_Trainer.ipynb:

This is a jupyter notebook that is used to train and test an image colorizer using our model. At the top of the notebook there is a cell with many of the global training option including batch size, number of training epochs, checkpoint directory, data directory, and the input size. The notebook contains its own batch function which reads in all of the images in the given data directory and outputs the LAB-colored L channels along with the whole LAB image as output. Additionally, the notebook will download and extract all of the necessary information that it needs including the pretrained-vgg-model as well as all of our trained models. You can select which model you would like to use as well as which loss function.

The notebook has been optimized so that it will continue to restore and train on a model if it finds one in the given checkpoint directory. The model will be saved / updated on each epoch which enables us to switch up some of the training options such as the batch size or learning rate throughout the training process. Furthermore, the training process outputs a graph of the training and validation error after each epoch of training. Finally, there is a function at the bottom of the notebook that will output some of the results on the model applied to the test set.

Colorizer_CompareModels:

This ipython notebook will take a variable number of images from the data set, run them through each of the three models and then show the images of the initial black and white image, the L2-loss reconstructed image, the Huber Loss reconstructed image, the pairwise mean squared error reconstructed image, and then the ground truth image. This file will also download all necessary files including the pretrained VGG model as well as our models.

Colorizer_OldPhotoRestoration:

This ipython notebook will read images from the given directory and run them through the model before outputting the reconstructed image using the model with the pairwise mean squared error loss function. This file will also download all necessary files including the pretrained VGG model as well as our models. Furthermore, the notebook has a boolean variable at the top that will resize the images if wanted. Otherwise, the model will output an image of the same dimensions as the inputted image.

Bibliography:

Dahl, Ryan. “Automatic Colorization.” *Automatic Colorization*, Jan. 2016, tinyclouds.org/colorize/.

Deshpande, Aditya, et al. “Learning Diverse Image Colorization.” *CoRR*, abs/1612.01958, 2016, dblp.uni-trier.de/rec/bibtex/journals/corr/DeshpandeLYF16.

Hwang, Jeff, and You Zhou. “Image Colorization with Deep Convolutional Neural Networks.” *Stanford University*, 2016, cs231n.stanford.edu/reports/2016/pdfs/219_Report.pdf.

Mao, Xiao J, et al. “Image Restoration Using Convolutional Auto-Encoders with Symmetric Skip Connections.” *CoRR*, abs/1606.08921, 2016, arxiv.org/abs/1606.08921.

Zhang, Richard, et al. “Colorful Image Colorization.” *CoRR*, abs/1603.08511, 2016, pp. 649–666., doi:10.1007/978-3-319-46487-9_40.

Code Sources / Inspiration / Tutorials:

https://www.tensorflow.org/

http://www.vlfeat.org/matconvnet/

https://github.com/umuguc/matconvnet/tree/autoencoder/examples/mnistAutoencoder

https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/autoencoder.py

https://www.learnopencv.com/understanding-autoencoders-using-tensorflow-python/

https://github.com/shekkizh/Colorization.tensorflow

https://jmetzen.github.io/2015-11-27/vae.html

http://cvcl.mit.edu/database.htm