Post-implementation report

Photo Colorization Application

June 24, 2020

kristofer bitney

WGU C964

Table of Contents

[Business Vision 2](#_Toc43899184)

[Dataset Artifacts 3](#_Toc43899185)

[Code Analysis 5](#_Toc43899186)

[Data Display Effectiveness 9](#_Toc43899187)

[Neural Network Results Analysis 9](#_Toc43899188)

[Application Testing 10](#_Toc43899189)

[Application Files and Installation Guide 11](#_Toc43899190)

[Server Setup 11](#_Toc43899191)

[Photo colorization 13](#_Toc43899192)

# Business Vision

We love our customers, and our customers love photos. Augmenting our photo restoration products and services, the photo colorization application will bring a new vividness to our customers’ memories. The application is easy to use and is almost entirely automatic. Importantly, marginal costs are low: colorizing an additional photo takes less than thirty seconds and cost nothing after monthly server maintenance costs are paid. We can now increase customer happiness by offering photo colorization at a low price, with little additional costs to our organization.

The application is hosted at https://imgbuff.onrender.com/ and can be accessed from desktop or mobile devices. Users are presented with a streamlined version of the application when accessing the website from a mobile device. The application is based on a neural network that takes grayscale images as input and outputs colorized versions of the images. The neural network is abstracted out of the way, so that users need not even know it exists. Users simply navigate to https://imgbuff.onrender.com/, select an image file, and click a button to receive a colorized copy of their photo.

# Dataset Artifacts

The dataset included about 118,000 color photos from the Common Objects in Context (COCO) dataset, available at https://cocodataset.org.

I used the Gradient platform to develop and train the neural network. The platform offers virtual machines that emphasize the use of Jupyter Notebooks for data science development. I downloaded the data directly from https://cocodataset.org and stored it in the persistent remote storage that comes with a Gradient subscription. The functions I used to download the data are pictured below. The functions chunk the data file into 100mb pieces to increase download speed.



I made a grayscale copy of each photo using Python’s Pillow library. The grayscale copies served as model input, while their color analogs served as “labels” to which model output was compared. I used the following functions to make the grayscale copies. The functions make use of the FastAI library’s utility function for parallel data processing.



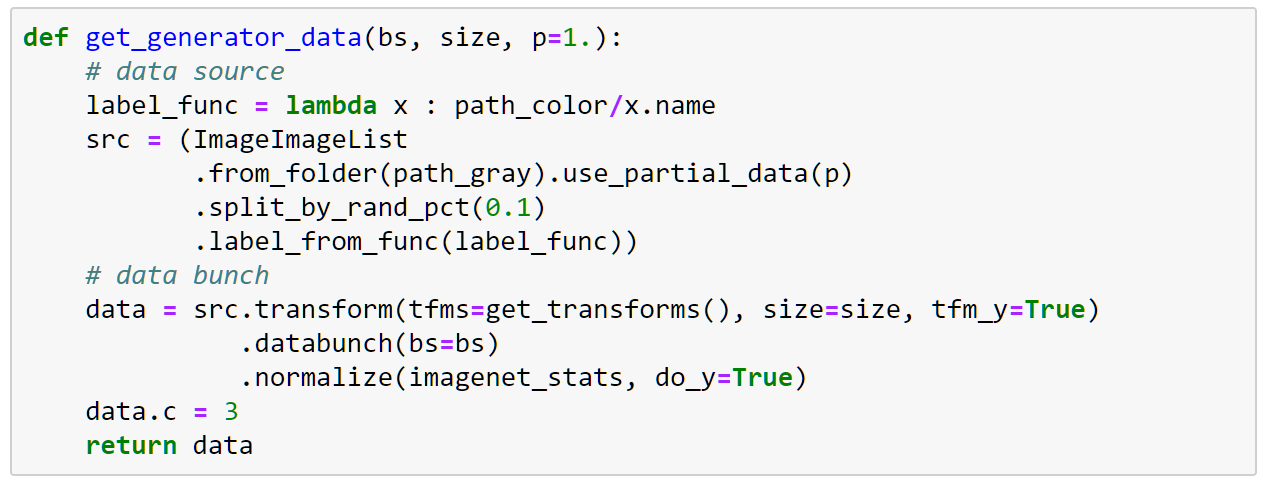
Here is an example from the dataset. The original image (right) was converted to grayscale (left) before being colorized by the neural network (middle).



It is important to note that I resized images to fit my neural network models. I began by resizing images to 128x128 and ended at size 320x320. By training the model at smaller image sizes before moving on to the target image size, I was able to effectively increase the size of the dataset. This is considered a best practice.

Randomized transformations were applied to the input photos and their labels in order to augment the dataset. This is another best practice used to increase the effective size of the dataset, improving the model’s robustness. The transformations included minor lighting adjustments, rotations, “flips”, zooming in, and warping.

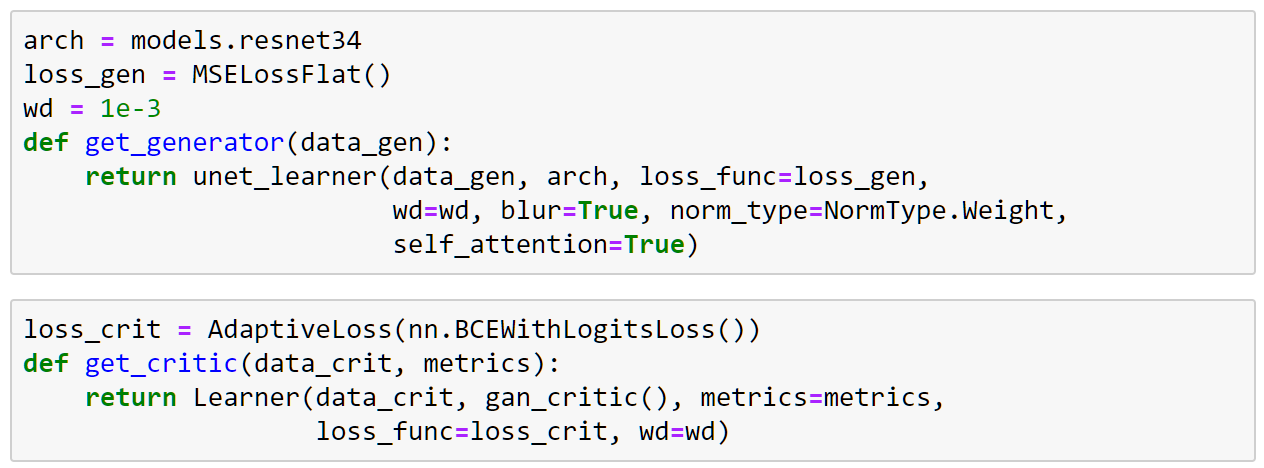
Image resizing and transformations are applied lazily by functions implemented as part of the FastAI library’s Data Bunch API. One of my applications of the Data Bunch API is pictured below. The API pulls grayscale images from the folder, splits about 10% of the training dataset into a validation dataset, and identifies label images using a lambda function. It also applies my specified transforms “on the fly” during training and normalizes the dataset.



# Code Analysis

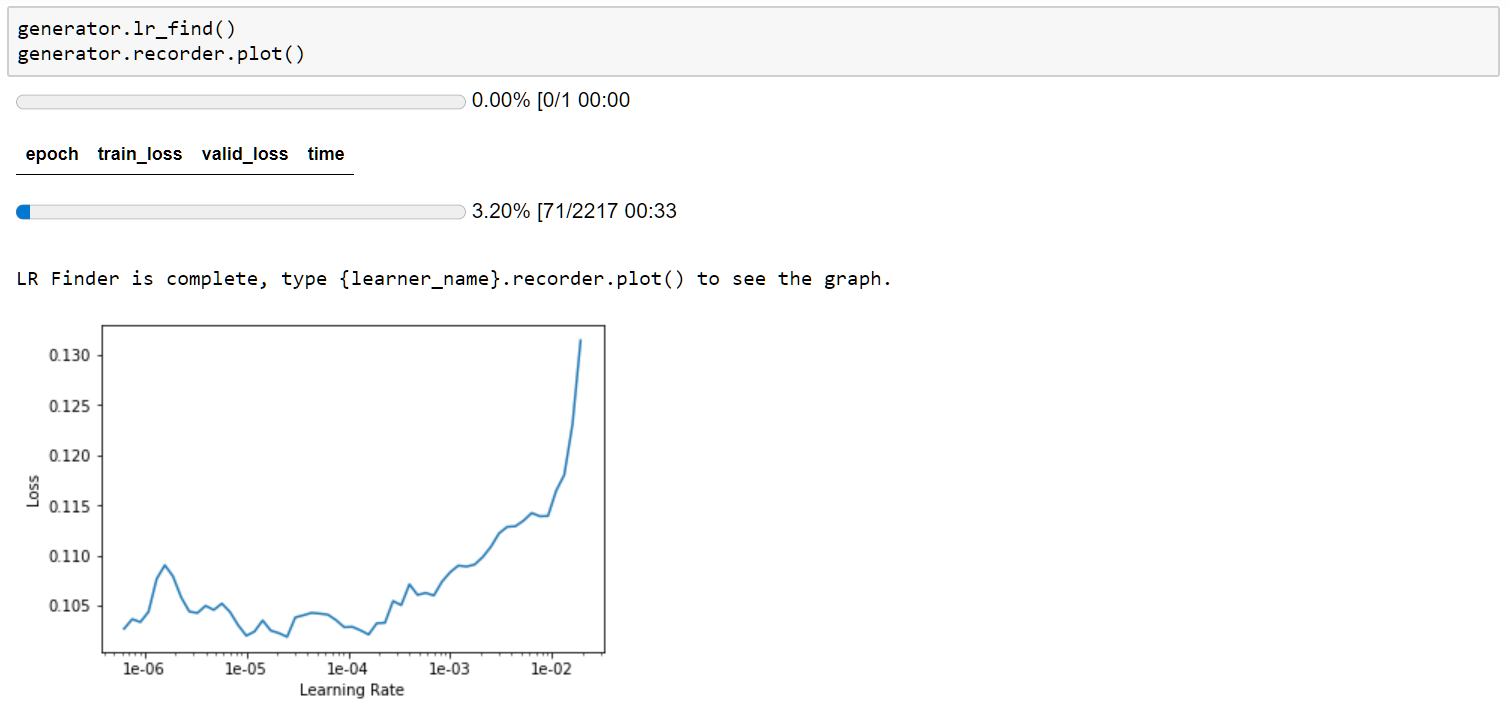
I primarily used the FastAI Python library to construct the neural network used in the application. FastAI reduces boilerplate code, allowing developers to focus on their deep learning models.

The functions I used to set up the “generator” and “critic” neural networks for my Generative Adversarial Network (GAN) are pictured below. The generator colorizes images, while the critic discriminates between generator output (colorized images) and their labels (the original color images).

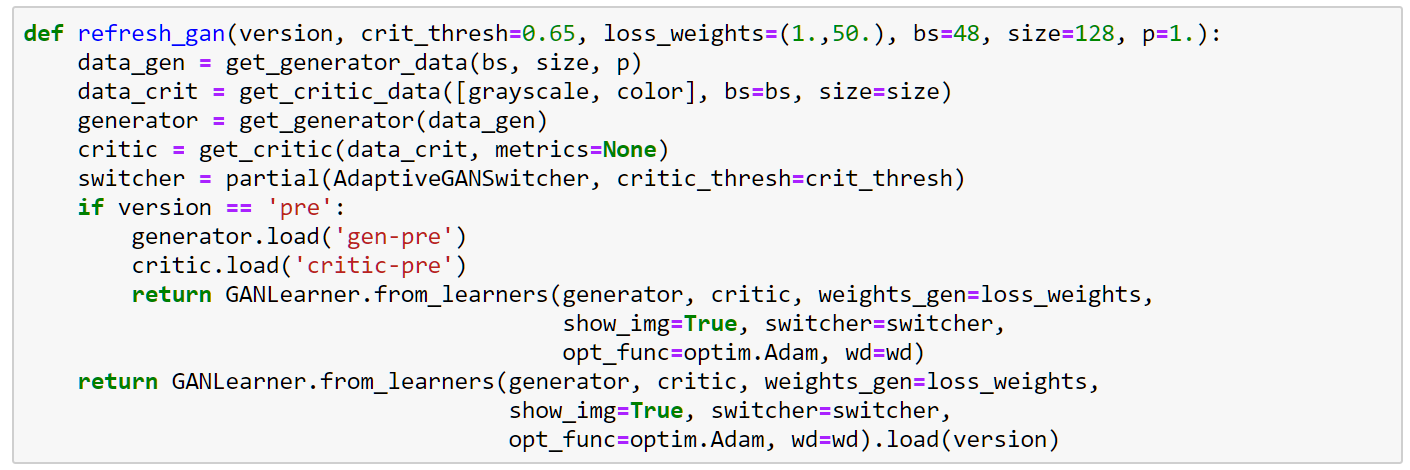


I first pre-trained the generative neural network and saved colorized versions of every image. I then pre-trained the critic neural network.

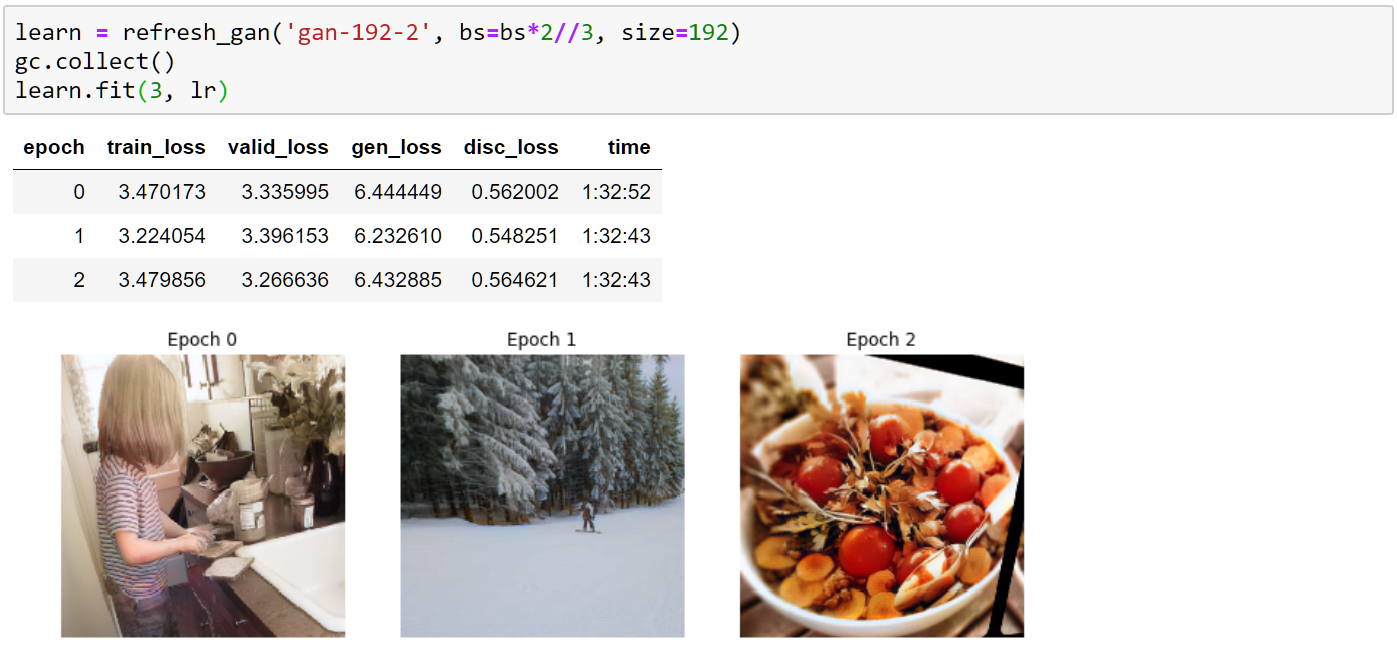
On occasion, between training iterations, I used the learning rate finder implemented by the FastAI library. The learning rate finder is based on empirical results reported in Smith L.N. (2017)[[1]](#footnote-1), which provide guidance on how to choose a learning rate. The learning rate finder estimates how each learning rate (within a range) will affect the model’s loss function.



I created a Generative Adversarial Network based on the pre-trained generator and critic networks. Because the Gradient Platform limits use of free GPU compute services to six-hour sessions, I created a function to load saved copies of the GAN. The final training process took more than two dozen six-hour sessions, not counting sessions used for experimentation and human learning.



The FastAI library’s training functions abstracts away from PyTorch’s training loop code. It also prints useful training information by default, so I was able to monitor training progress. I set the GAN to print an output image after each epoch. This made it possible to observe improvements in the end product while the model trained. Some training output is pictured below, with results that looked particularly promising.

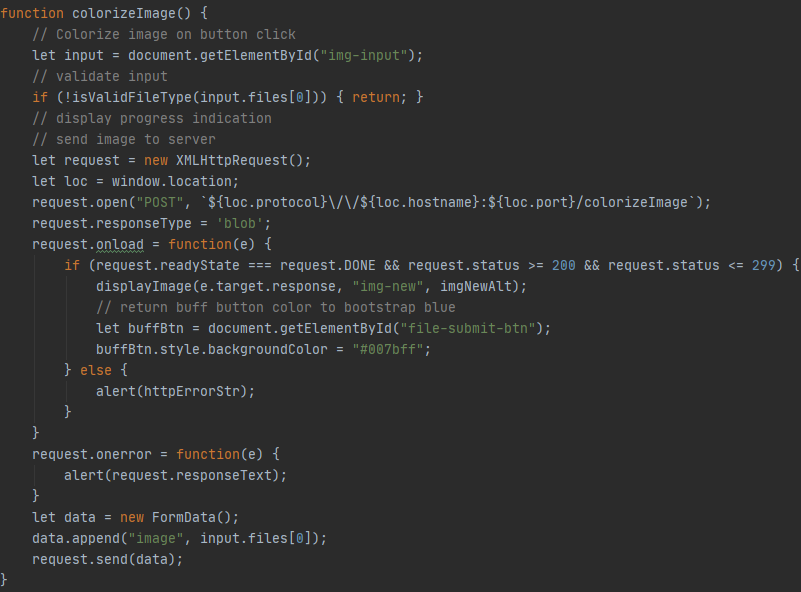


Once the GAN was trained, I uploaded the network directly to DropBox account I created for the application. While I was searching for solutions, I found a useful function posted by a user on StackOverflow. I copied and modified the solution, and cited my source in the code comments.

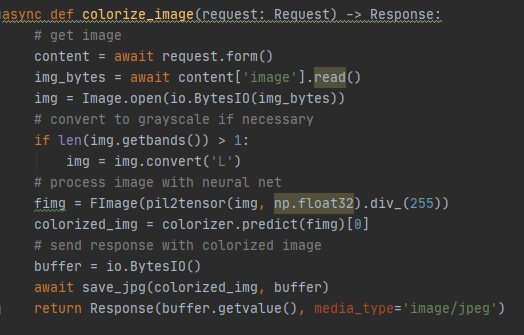


I also wrote a website using HTML5, CSS3, and JavaScript to make the application accessible to non-technical users. The website provides users with a graphical user interface. Users can colorize photos with only a few mouse clicks. If it were not mentioned explicitly on the website, users would not know that neural networks or web servers are involved at all.

When a user submits an image file on the website, the following JavaScript function is called. The function creates an HTTP request and sends the image file to a web server.



The web server runs a Python module I wrote using the Starlette framework for Python. Starlette implements the Asynchronous Server Gateway Interface (ASGI) standard, making it possible for the server to handle asynchronous events. While the full back-end script is more complex, the core function written to handle colorization HTTP requests is pictured below.



# Data Display Effectiveness

The most robust way to assess the performance of a neural network is to review its performance on a validation dataset. The loss estimate of a statistical model on a validation dataset is a measure of Expected Prediction Error—i.e., a measure of how the model will perform on real-world data that it has never seen before. I used direct measurements of model loss to evaluate the performance of the generator and critic neural networks while pre-training them.

Unfortunately, validation loss is not a useful measure of performance when training Generative Adversarial Networks. As the generator network learns to fool the critic network—by improving its colorization performance—its loss decreases. However, the critic receives additional training as the generator’s loss decreases. This makes the critic harder to fool, increasing the loss of the generator. In other words, the validation loss of the generator fluctuates during training but does not meaningfully decrease.

Instead, the best indicator of GAN performance is qualitative observation of the generator output. The FastAI library makes this easy. As pictured in the Code Analysis section of this document, the FastAI library prints an example output image after each training epoch (iteration). I monitored the output, looking for signs of under-fitting and over-fitting.

Note that data preparation is also discussed in the Code Analysis section of this document.

# Neural Network Results Analysis

The final results of the neural network are often very impressive, and difficult to distinguish from real color photos at first glance. But when compared to their original counterparts, two flaws are sometimes visible.

The first flaw is under-colorization. The model tends to err toward duller, less vibrant colors. The original color photos tend to have more vivid, brighter colors. Sometimes the model does not colorize parts of images at all. For example, it might make a car white when the car is yellow in the original photo. The second flaw is discoloration. The model sometimes applies color where it should not be. Discoloration is somewhat rare, but tends to stand out when it occurs.

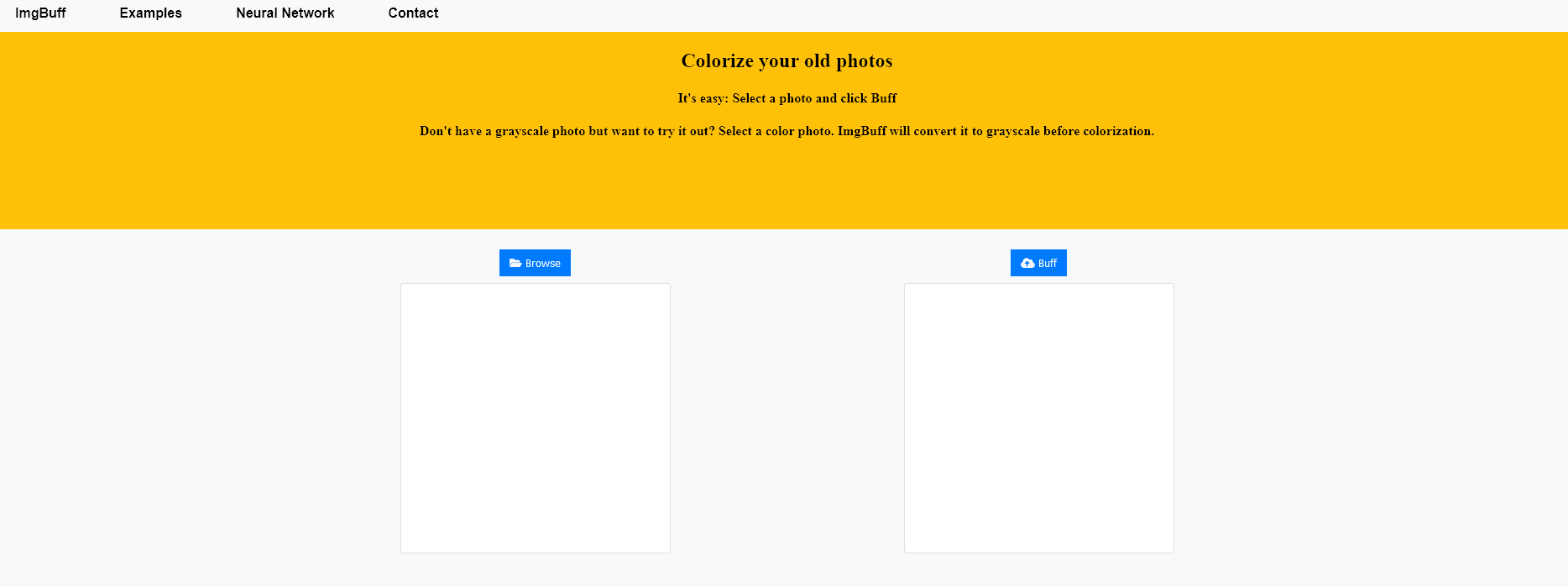
In the images below, the colorized photo (middle) has slight bluish discoloration at multiple locations and less color contrast in general. The original color photo (right) appears more vivid.



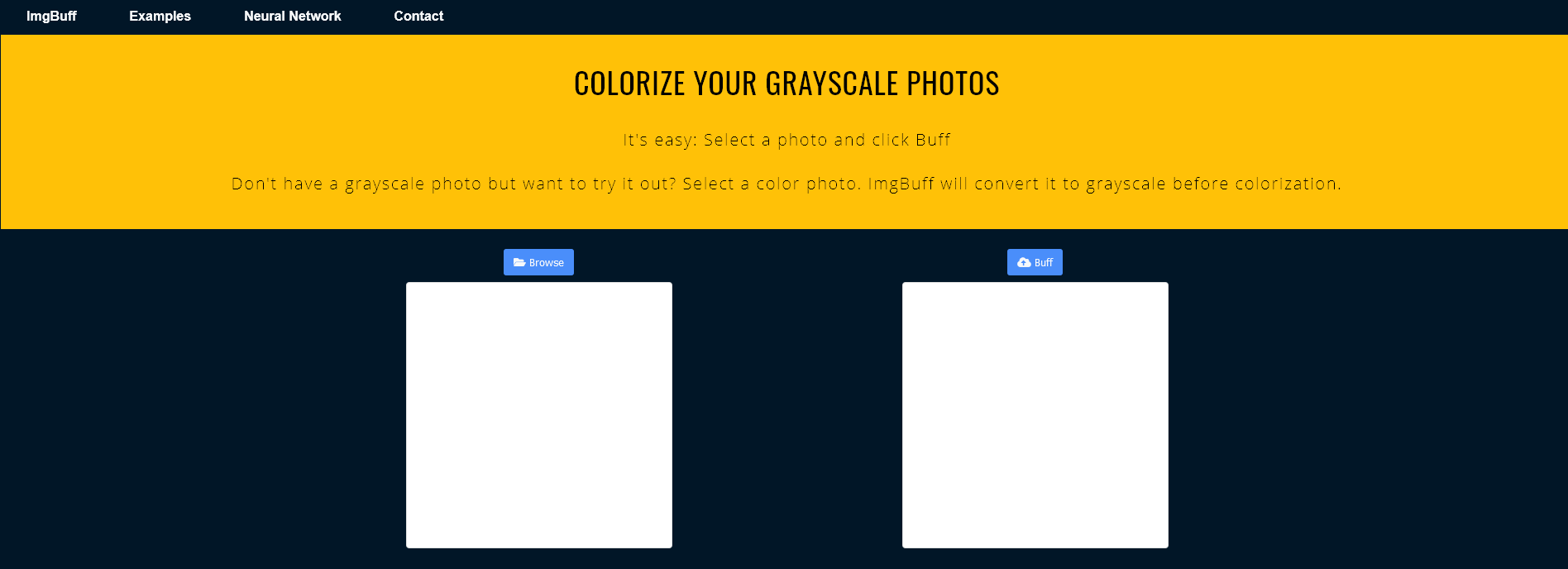
# Application Testing

The primary form of testing I used was usability testing. I created a questionnaire—see Appendix D of the Technical Summary document—and requested assistance from friends and family. Usability testing is the most important form of testing for the photo colorization application because the application’s value primarily depends on users’ perceptions of the quality of its colorization results. Usability testing helped me assess whether users found it easy to colorize their photos, did not encounter errors, and were pleased by the results. Usability testing also guided website design refinements.

Below is a screenshot of users’ first look at the website, before usability testing.



After testing, the landing section of the website was refined to increase color contrast and improve image visibility:



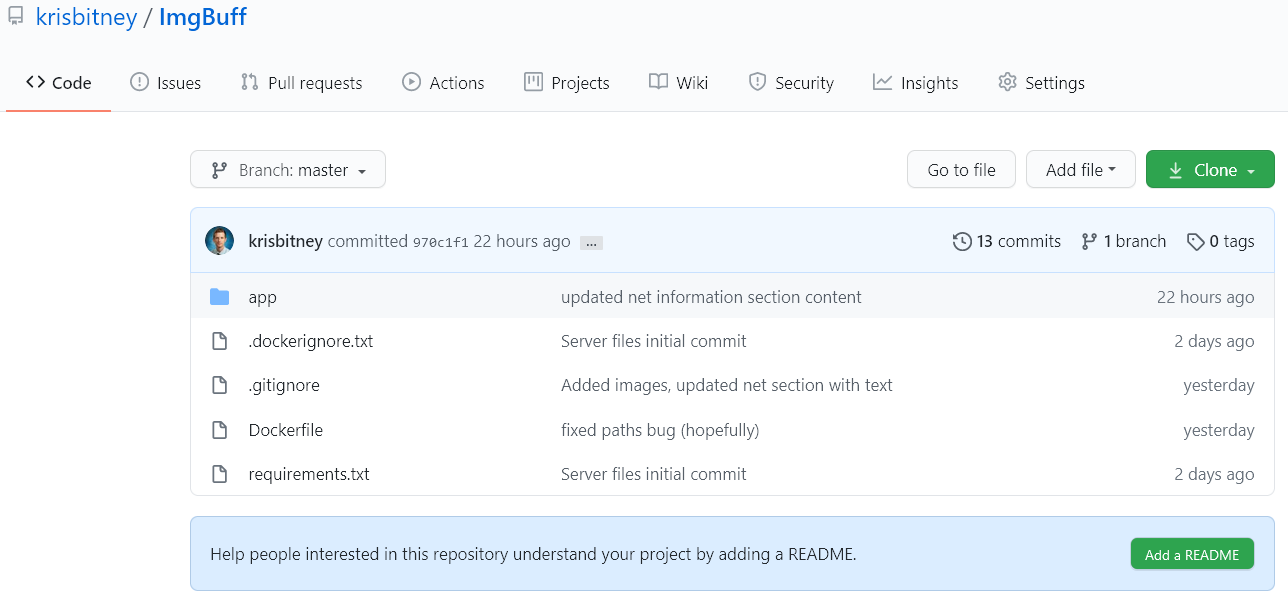
Other changes influenced by testing included the arrangement of website elements, fonts, and text sizes.

# Application Files and Installation Guide

End users can access the application by going to https://imgbuff.onrender.com/. No file installation is necessary, but an internet connection is required. If you simply wish to use the application on that website, skip to the Photo Colorization section at the end of this guide.

## **Server Setup**

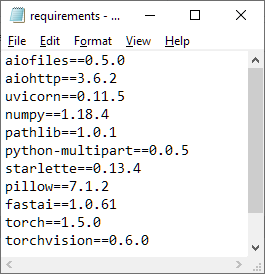
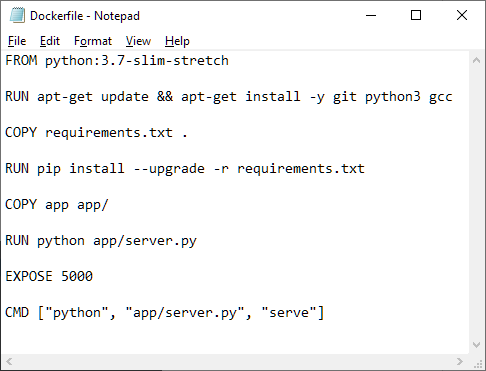
The initial process of hosting the application on a web server is slightly more complicated. To host the application on Render, where it is currently hosted, the application files need to be pushed to an otherwise-empty GitHub repository. The easiest way to accomplish this is to fork the existing application repository from <https://github.com/krisbitney/ImgBuff>, pictured below.

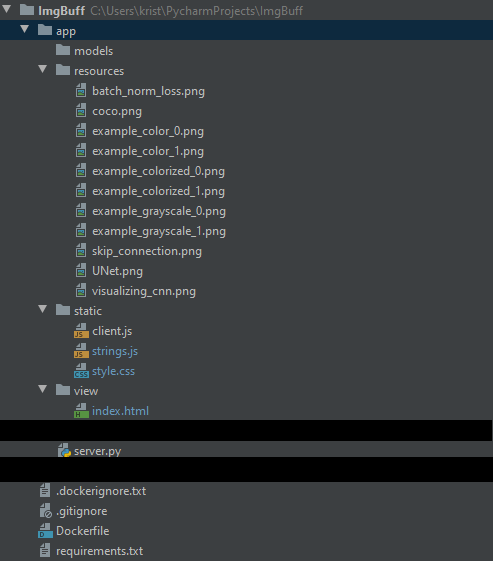


Should you wish to create your own repository directly, the file structure of the application should be maintained as pictured below.

Note that the neural network file is downloaded by the web server, is not part of the file structure that should be uploaded to GitHub, and is not part of the project submission. The server.py module contains the code used to download the network. This approach is necessary because the serialized neural network file exceeds the GitHub file size limit of 100mb.

The “Dockerfile” and “requirements.txt” files provide the server instructions to launch the server.py module—which downloads the neural network file and manages HTTP requests—and install project dependencies when the application is deployed on the web server. The requirements file lists the project dependencies, which the Docker build file tells the server to install with a Linux terminal command to the pip package manager. After issuing a command to install project dependencies, the Docker build file runs the server.py module which takes care of the rest. The contents of the requirements.txt and Dockerfile files are shown below.



An important Python module called “learn\_nn.py” is not included in the picture. It was added as a hotfix in the most recent update. It should be included in the “app” folder, alongside “server.py”. It changes the way the neural network is loaded by the server. It does not change the installation process.

Additional files included in the WGU project submission are WGU\_Capstone.ipynb and WGU\_Capstone.html. These are Jupyter Notebook files containing neural network training code. They are important for modifying the neural network but are not necessary for installation.

Once the project file structure is uploaded to GitHub, you can deploy the web application on Render with the following steps:

1. Create a Render account and log in.
2. Create a new Web Service.
   1. Associate the account with your GitHub repo.
   2. Render will request read permissions, which you must grant to use the service.
3. On the deployment screen, select Docker as the server build environment.
4. Click save.

It will take a few minutes for the server to deploy the application. Render observes the GitHub repo you used, which should contain the application files. When the repo changes, Render automatically re-deploys the application so it is always based on the latest version.

Render will automatically provide you with a free URL based on the name you gave the Web Service. You can test your deployment by visiting your URL. The server automatically includes a free SSL certificate for encrypted HTTP communication, so user data is secure.

## Photo colorization

To colorize a photo, click the “Browse” button in the middle of the page. You will be prompted to select a file. Select an image file. Note that if you select a color image file, the application will convert it to grayscale before colorizing it. Once you’ve selected a file, the folder icon on the “Browse” button will close and the “Buff” button in the middle of the page will turn green.



Once you have selected an image and the buff button has turned green, click the “Buff” button. You will be presented with the colorized copy of your photo, which you can then save to your local machine.

1. Smith, L. N. (2017). Cyclical Learning Rates for Training Neural Networks. ArXiv:1506.01186 [Cs]. http://arxiv.org/abs/1506.01186 [↑](#footnote-ref-1)