HMC CS 121 Fall 2020 **Final Report**
Project: Image Colorizer
Developers: Bernhard (MD), Shah (E), Ranabhat (K), Croom (MJ)

1 Abstract

Our Image Colorizer web app allows users to make a black and white image into a full color image. The front end is a pure HTML, JS, CSS website served by a Flask app and deployed with Gunicorn on a Heroku server. The backend is a convolutional neural net trained on the ImageNet database with 8 convolutional layers based on a project made in 2018. The pre-trained Caffe model is integrated into a Flask backend with the OpenCV python library and hosted on PythonAnywhere. Given the lightness channel of a gray-scale image, it predicts the color channels in order to create a colorized image that we deemed convincing to the human eye in most cases, satisfying our goals for the project.

2 Introduction

Color photography was not invented until the 1960s. For the century prior, individuals were only able to take black and white photos of people and the spaces around them. Since we lack the exact colors of such pictures, we are unable to truly recreate the past, at least to the extent that we are able to with modern photography. Although experts in image editing can colorize images by hand, the average person would benefit from having an automated service to colorize any black and white image they have. Our project seeks to provide users with a way to add relevancy to old photographs by creating a prediction for what the human eye might have seen during that time period.

There are a couple algorithms that solve this problem, but most of them require some human decisions, resulting in questionable results. Since these algorithms were human dependent, they were unable to consistently create a uniform output, resulting in the prediction and image quality being dependent on a flawed human decision process. One of the most promising algorithms that operates without human intervention is a convolutional neural net. Several neural nets that utilise CNNs have been developed over the years; the one we chose to use was developed by Richard Zhang and his research team in 2016. The model is pre-trained on the ImageNet data set. Given a black and white image, it predicts the colors from the light intensity. We deployed our frontend app on a Heroku server and used a Flask backend on PythonAnywhere, a python hosting service, to process requests with the neural network model. Our web app allows the user to upload an image to the model and colorize it. After the colorization process, the user can download the image to their local file system.

In the background section, we discuss some of the preexisting work on this problem. We discuss how we decided on the model we ended up using, and where this project stands compared to the current options for image colorization. In the functional requirements section, we describe how the app functions and what kind of input and output to expect. In the architecture section, we provide deployment, class/module, and sequence diagrams as well as descriptions of how the code functions and is deployed. In the algorithms section, we talk about the neural network model, how it works, how it was trained/developed, and its performance. In the testing section, we talk about our user tests and code tests, the feedback they generated, and how we improved the app based on this feedback. In the limitations and future work section, we explain some limitations of the app and talk about potential improvements going forwards.

3 Background

Our project is based around the idea that we want to use a neural net to predict what a black and white image would look like if it had color in it. We needed to use a neural net to address this problem because there is no definite answer or solution to this problem. To create this colorized image we are dependent on patterns arising from color photography that we can then run through a neural net, which can later create predictions for our black and white images.

The first paper we found on image colorization is titled Deep Colorization and was written by Zezhou Cheng, Qingxiong Yang, and Bin Sheng. As they describe, before their project, image colorization was unable to be fully automated due to anomalies that required a human perspective to take them out of the image. The most convincing algorithms involved having a human manually provide some colors for certain regions of the image or carefully hand select reference images depicting extremely similar scenes. The paper showed that using deep learning was significantly more reliable than any preexisting algorithm with a manual component used for image colorization as of 2015. It has previously been established that convolutional neural networks perform very well on image-based problems, so we came to the conclusion that a Convolutional neural network (CNN) was the best way to move forward.

For our project, we found a pre-trained CNN with promising results created by Richard Zhang in 2016. The algorithm implemented in their paper used a feed-forward neural network that allowed all information to move through a series of convolutional layers in a one direction only. The CNN is structured so that the image must go through 8 convolutional layers, ensuring that we make enough predictions on the color that we can find the most accurate version of the colorized image (see Figure 1). The model that we are using is publicly available as a Caffe model. This model was pre-trained on ImageNet and has created predictions it can use to guess the color presence in black and white images.

However, the model was only a small part of creating the backend. The most important part was this web page. This tutorial helped us write the code for our backend. This CNN extracts the LAB color space from the image, instead of the traditional RGB. In the LAB color space, the L channel represents Lightness, the a channel represents the green-red spectrum, and b channel represents the blue-yellow spectrum. So to add colors to a black and white image the neural net uses the Lightness information given, puts it through the Caffe model several times, and then outputs a prediction for the a and b channels.

Additionally, we chose to use the OpenCV platform to help perform our predictions. This allowed to us to easily read and process the images we wanted to put through our CNN. OpenCV provided us with a reasonable space to analyze and manipulate our image, and allowed us to return a digital image that was colorized.

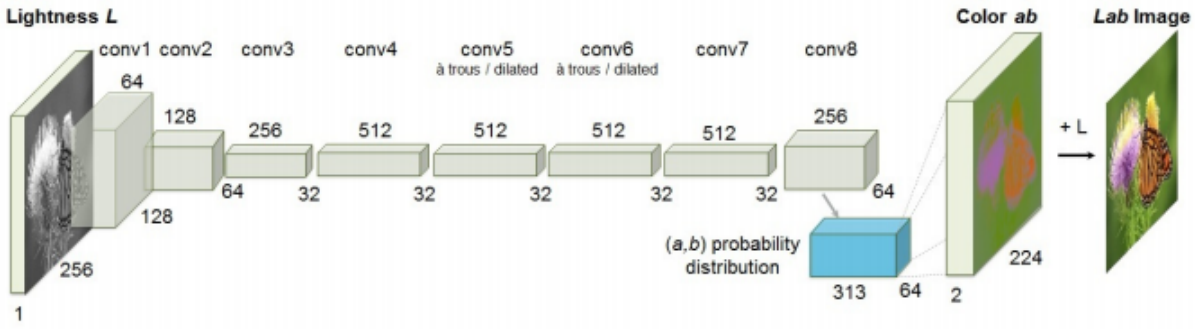


Figure 1: Our network architecture. Each **conv** layer refers to a block of 2 or 3 repeated **conv** and **ReLU** layers, followed by a normalization layer. The net has no pool layers. All changes in resolution are achieved through spatial downsampling or upsampling between conv blocks

The frontend is a simple php app that serves a basic web page created with HTML, JS, and CSS. The web page features three tabs, a "Colorizer", "FAQ", and a basic "About" page that describes the project. On the colorizer tab, users can upload an image from their local machine, view it, and then elect to replace it or colorize it. When they opt to colorize it, the image is sent via a POST request to the Flask backend on PythonAnywhere, which colorizes the image using the neural network then returns the results in the response. The result is then displayed to the user on the frontend. They can then elect to remove the image or download it.

4 Functional requirements

Our app is designed to allow users to colorize black and white photos that they have either found on the internet, or their own black and white photos that they have taken. This process should be relatively simple and easy to use.

The user can access our app at the main page.

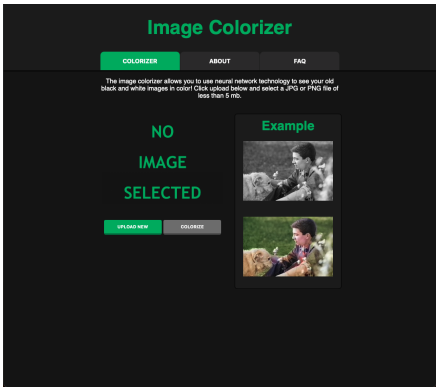


Figure 2: Main page of app.

When the user opens our app they will see the main page as shown in Figure 2. They will have a few options from this point. At this point they can begin uploading an image, learn a little bit about our app, or look at our Frequently Asked Questions (FAQ) page. If the user wants to upload an image they simply press the upload button.

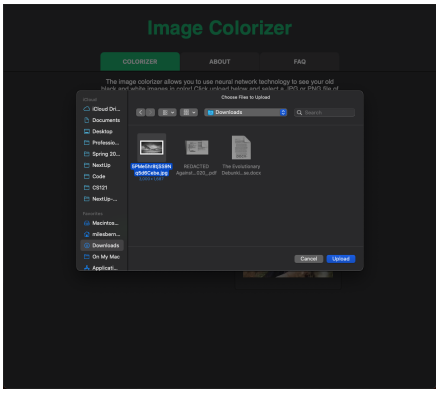


Figure 3: Upload selector.

After pressing the upload button, the user will be taken to a screen where they will be asked to pick an image from their computer and upload it to our app, as shown in Figure 3. At this point the file browser will only allow you to select jpg and png images to upload to our app. Our app will take in either a jpg or png image that is under 5 MB. Preferably, this image would be black and white, however our image colorizer will also take in colored images. This alternative entry will run the grayscale version of your colored image through our network and create a prediction, which you can compare to your original photo.

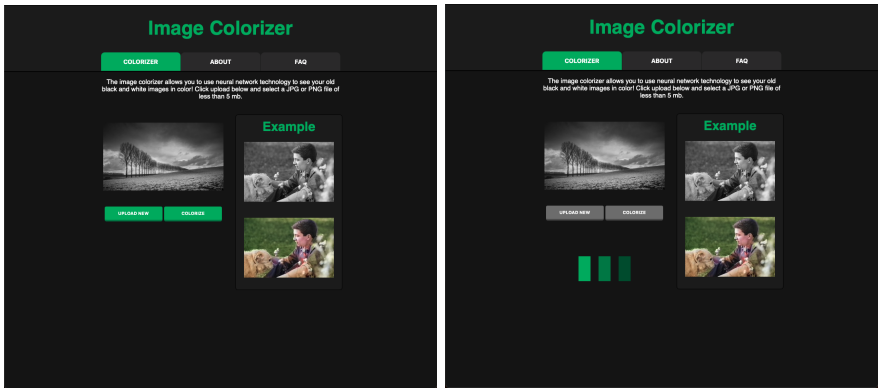


Figure 4: a. Image Uploaded before colorizing. b. After pressing colorize button.

The app will attempt to upload the image. If the image is too large, an error message will show, and the user will be asked to upload a new image. If the app is able to upload the image the user will be taken to the screen shown in Figure 4.a. If they are happy with the image they uploaded they can press the colorize button and proceed with colorizing their image. If the user is unhappy with what they uploaded for whatever reason, they can then press the upload new button and pick another image to upload to the app.

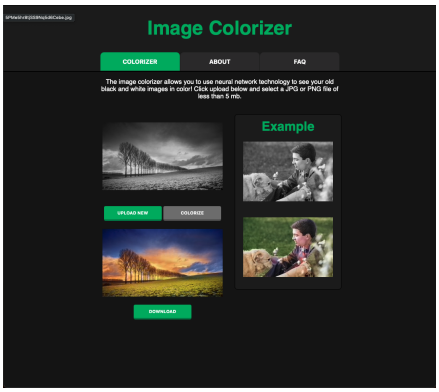


Figure 5: Colorized image loads.

Once the app has completely colorized the image, the app will return an image to the main screen. At this point, the user has the option to start over with the image colorizing process or to download the result of the colorizer. By simply pressing the download button, the image produced will be stored locally on the user's computer for later viewing. The output will always be downloaded as jpeg file.

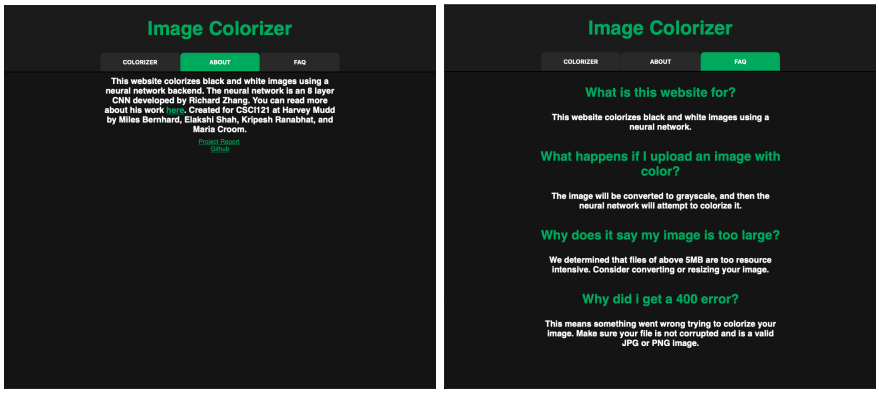
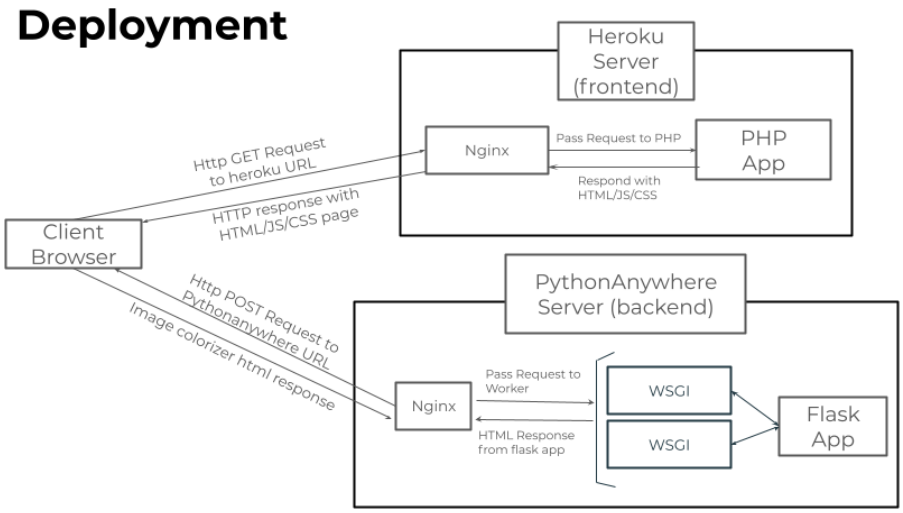


Figure 6: a. About tab. b. FAQ tab.

In addition to colorizing an image, there are a couple of other tabs the user has access to. All of these tabs can be accessed along the top bar of the app. As shown in Figure 6.a, the About tab has some basic information about the resources we used and has links to our final paper and code (through GitHub). Figure 6.b shows the FAQ tab, where the user can get answers to questions that they might have.

Overall, this app is intended for to be very accessible to users. We want users to be able to navigate through the app easily without many obstacles. We aided this goal by creating the About and FAQ tabs, so that if the user is confused, they have several ways to learn more about the app and our project.

5 Architecture



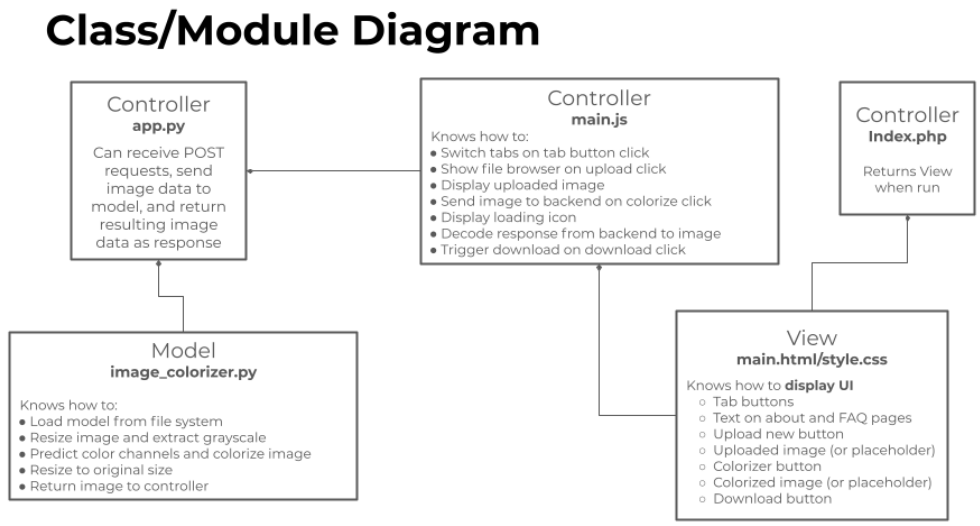


Figure 7: FAQ tab.

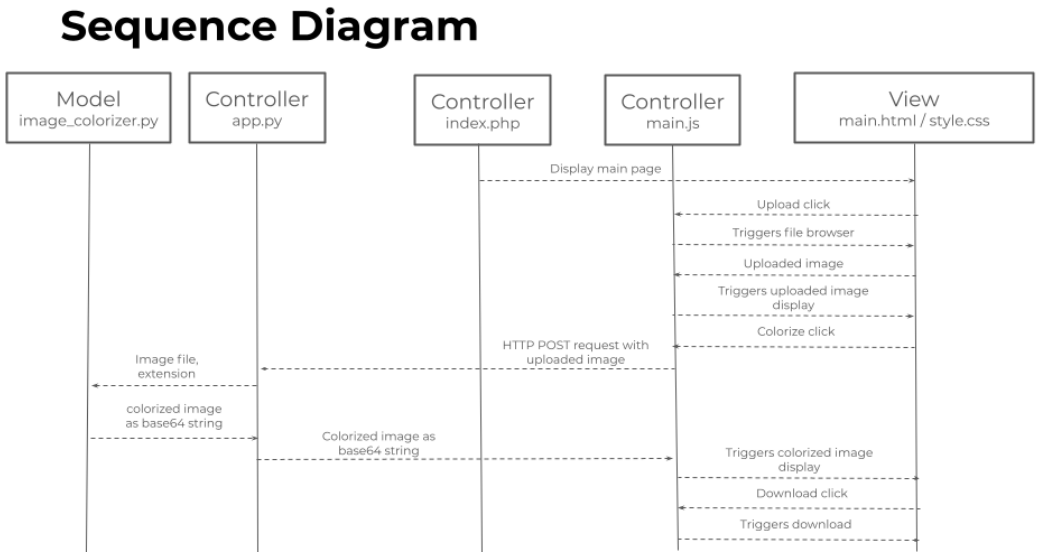


Figure 8: FAQ tab.

The frontend of our application is a basic PHP application that simply serves an HTML, JS, and CSS application at the route directory. We opted for basic HTML, JS, and CSS as the frontend of the application is not super complicated and didn't require something more complicated like React or Angular. We used PHP as we found it was the simplest way to run HTML, JS, and CSS as an application for deployment on Heroku. When deployed on Heroku, the server uses Nginx to handle receiving and passing requests to the PHP app and returning the response to the users browser. See the deployment diagram and class module diagram above.

The backend of our application is a Flask app. We chose to use Flask and Python because we are most familiar with Python and found Flask to be the simplest way to get a backend up and running. The

backend is deployed on PythonAnywhere, which is a simple and cheap hosting service for Python code. When deployed, the server receives requests with Nginx and passes them on to WSGI workers that interface with the Flask app to generate a response to send back to the client. Again, see the deployment diagram and class module diagram above for a visual representation of this flow.

On the frontend, we use buttons to manage the tabs. Each button shows or hides various divs to represent changing tabs. The HTML code points the buttons to the JS function 'openPage' which takes a page name as an argument. It then changes the CSS on all the tabs to hide them and set the respective button as not selected using CSS, other than the tab passed in the argument which is shown and has its button highlighted. We opted for this single page solution since it is simple and allows the user to change tabs without losing progress on the colorizer tab, since the elements are all still present, just hidden.

The main user flow comprises of uploading, colorizing, and downloading an image. Upon clicking the upload new button, the uploadClick function is called. This hides the return image and download button, then displays an image selector to the user. Once an image is uploaded, the JS function confirms the image exists and is displayed correctly, and otherwise shows an error. If all goes correctly, the colorize button is enabled (this is represented by changing it from grayed out to green). From there, the user can click Upload New again (restarting this sequence), or colorize. When colorize is clicked, the JS function colorizeImage is called which displays the return image as a loading gif and disables all visible buttons, then converts the uploaded image to a file and sends a post request to the backend endpoint. Upon receiving a result, it confirms the result is okay. If not, it hides the result image and displays the error message recieved from the backend. If it is okay, it converts the base64 encoded result to a file then sets the return images source to be this resulting image to end the loading graphic and show the user the result. It also enables the download button. Finally, the user can click download, which triggers a download to the users local machine with the filename formatted as originalname-colorized.extension. From there, the user can only download again or upload new. This sequence is represented by the sequence diagram above.

The backend application consists of one endpoint, /colorize. This endpoint recieves POST requests only. When the Flask app is initialized, the app.py file is run. This file imports the image_colorizer.py file, which triggers it to run. When it runs, it loads the neural network file from storage using the OpenCV library. When the app receives a POST request to /colorize, It tries to extract the image file from the requests and parse the extension of the image. Then, it confirms the extension is acceptable. If it is not, it raises an error. Otherwise, it calls the colorize_file function from the image_colorizer module with the file and the extension. This function converts the image to LAB, extracts just the L channel, and resizes it to 224 by 224, which is the dimension accepted by the network. Then, it uses the neural network to predict the A and B (color) channels of the image. The resulting channels are synthesized with the original L channel and scaled back to the original image size. The result is encoded as a base64 image and returned to the app.py file, which returns the result to the requester with the image and a 200 status code. If, at any point, an error occurs, it is caught and the app returns an error status code along with a message representing the error. This flow can best be seen in the sequence and class module diagrams above.

6 Algorithms

The neural network is the only non-trivial algorithm used in this project. Our model was pre-trained, which means that we personally did not train the model, but we will still include how the model was trained by Richard Zhang and his team.

The CNN was trained to map a grayscale input to a distribution over quantized color value outputs using 8 convolutional layers. The network uses its own loss function because the researchers found that the natural objective function, Euclidean loss L_2, is not ideal for handling the ambiguous and multi-modal nature of colorization. So, they implemented a multimodal cross entropy loss function and mapped a probability distribution to the various color values. At that point the results were not colorful enough, as the distribution of ab values in most images tend towards the low values of ab because of the appearance of clouds, pavement, walls, etc. To handle this, Richard Zhang added class rebalancing module that reweighs the loss of each pixel during training based on the rarity of the pixel's color, where the rarity is determined by the frequency of that color in the full ImageNet training set. The distribution between colors then is smoothed out with a Gaussian kernel $\mathbf{G}_\lambda$ with $\lambda = 5$.

Zhang and his team trained this network on the 1.3 million images from the Image Net training set. They then proceeded to validate their results on the first 10 thousand images in the Image Net validation set. They did one more test on a separate 10 thousand images from the Image Net validation set. They trained the model from scratch with k-means initialization and used a solver for 450k iterations. Furthermore, some versions of the model were adjusted using various tuning and balancing methods.

Colorization Results on ImageNet

Method	Model Params (MB)	Model Feats (MB)	Runtime (ms)	AuC non-rebal (%)	AuC rebal (%)	VGG Top-1 Class Acc (%)	AMT Labeled Real (%)
Ground Truth	–	–	–	100	100	68.3	50
Gray	–	–	–	89.1	58.0	52.7	–
Random	–	–	–	84.2	57.3	41.0	13.0±4.4
Dahl [2]	–	–	–	90.4	58.9	48.7	18.3±2.8
Larsson et al. [23]	588	495	122.1	**91.7**	65.9	**59.4**	**27.2±2.7**
Ours (L2)	129	127	17.8	91.2	64.4	54.9	21.2±2.5
Ours (L2, ft)	129	127	17.8	91.5	66.2	56.5	23.9±2.8
Ours (class)	129	142	22.1	91.6	65.1	56.6	25.2±2.7
Ours (full)	129	142	22.1	89.5	**67.3**	56.0	**32.3±2.2**

Figure 9: Table 1 from Zhang. Demonstrates various metrics that determine the success rate of different algorithms. Variants that are important to note are the L2, fine-tuned L2, and class rebalanced variants.

To evaluate the success of the training completed, the researchers also performed a series of 3 experiments to determine the success rate of their models.

The first metric used was the Perceptual realism experiment run on Amazon Mechanical Turk (AMT). This experiment pitted colored images versus re-colorized images that had been put through a colorizing algorithm. The basis of this experiment was the human eye, as ultimately the measure of success for this algorithm was determined by an individual's ability to spot the difference between the two images. In the experiment, participants were shown two photos and asked to click on the photo they believed to contain fake colors. Several checks were put in place to ensure success, and ultimately they found that the algorithm fooled participants in 32% of trials, a higher amount than comparable algorithms.

The second metric used was a Semantic interpretability experiment run using VGG classification. This experiment determined if a generic object classifier, one that might have been used on Image Net images, could correctly classify the generated images. The experimenters determined that when color was taken out of the image, colored image turned to black and white, performance of the classifier dropped from 68.3% to 52.7%. When they colorized the black and white image they were able to achieve a performance accuracy of 59.4%. While Zhang's algorithm did not achieve the highest level of performance among all the compared algorithms, it did show an improvement when compared to the classification performance for balck and white images. This data suggests that it is for the classifier if the image is colorized, and thus provides a concrete use for the colorizer developed.

The last metric the research team used was data from a raw accuracy analysis. This analysis was fairly low-level, as it relied on numbers to determine the performance of the algorithm, whereas the researchers were more concerned with plausibility. The researchers computed the percent of predicted pixel colors within L2 distance of the ground truth (original image) in the ab color space. This produced a cumulative mass function which was integrated under the curve and normalized. The raw accuracy analysis determined that there were some difficulties optimizing the network from scratch, but otherwise the network performed accurate colorization.

7 Testing

7.1 User Tests

The user test were extremely helpful and we got a lot of useful feedback concerning our interface from the tests. Each member of the team conducted a user test by having the tester click through a wire frame of our interface. We asked them questions and observed how they interacted with the interface, eventually deciding on a few essential changes we needed to make. The most important one was fixing the title. Originally the title was *Black and White Image Colorizer*, but this actually mislead most of our testers who believed our app would make an image white and black, which was not what we were doing. We decided to change the title to *Image Colorizer*, which might be less detailed, but is significantly clearer for the user.

Another comment our testers had was that they did not totally understand what the error messages meant. This lead to the creation of an FAQ tab, that answered questions about why the user was not able to upload a certain type of file or why they were getting an error message. We also decided to add more information in the About tab to help users understand more about the frontend of our app.

One piece of feedback we received but decided not to change was the idea that once a user uploads an image, the app should automatically colorize the image and give the user the output. We decided against incorporating this into our app, because we wanted to insert extra checks for the user to make sure they selected the right image.

In the user tests we also asked how this app could be better structured so that a user might be more inclined to use it in their daily life. One of the most common responses was that the tester said they would use it more if the speed of colorizing increased. We took this feedback and directly applied it to our app, changing the architecture structure in our minimum viable product to one that worked faster and was more elegant. It was also suggested that we add examples to our main screen so that users could get a better idea of what was happening with our app. The examples on the main screen can be seen in the previously shown Figure 2.

7.2 Code

We started by looking at potential frontend issues. We weren't successful at trying to upload files with extensions other than .jpg or .png. The browser upload window itself limited our choices to just .jpg and .png files. If the file we are trying to upload is too big, we can get an error message (Figure 3a).

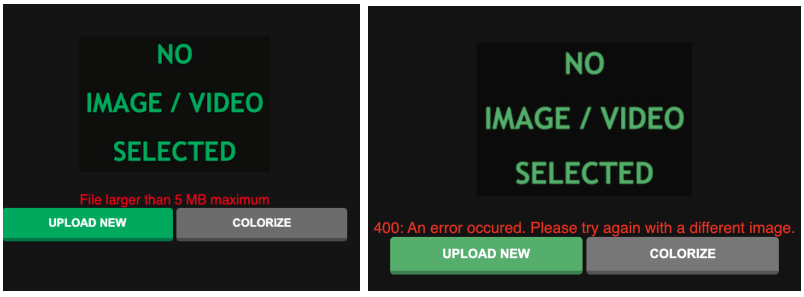


Figure 10: (a) The error message when the file is too big. (b) The error message if the file is an incorrectly formatted image.

We ran into an issue where when given an altered image file with the correct extension (which in our case was originally a text file renamed as a jpg file), our web app would hang and not let us upload a new image or colorize it. We determined by looking at the console output that the backend was correctly returning an error message, but the frontend was crashing trying to display it. The result was that no error message was displayed, and the upload new button was actually disabled despite appearing enabled. We fixed the code that was causing the error display to crash, which resolved the issue. Now, the user will see an error if they try to colorize an incorrectly formatted image file (Figure 3b) and will be able to upload a new one. We then determined that the frontend should just check if the image loads correctly in the first place rather

than sending it to the backend and receiving an error in response. So, we implemented an onLoad function that confirms the image loaded as intended and otherwise displays an error and removes it.

On the backend, we had some potential vulnerabilities. First off, while we check the image size and type on the frontend before posting it to the backend, we don't check it again on the backend. This means someone could theoretically make a request directly to the backend with an image that is too large or the wrong type. This would potentially cause the backend to crash. Although we have cross-origin protections on the backend, so it only allows requests from the heroku url or from localhost, someone could still make requests from localhost or spoof the origin of their request to our heroku link with a basic proxy setup using something like nginx. So, we implemented check on the backend to confirm the file type and size. The file type check was done by parsing the name of the file and comparing it to an acceptable list, and the file size restriction was achieved using the MAX-CONTENT-LENGTH attribute of the app's configuration.

In general, all calls to the image-colorizer file are wrapped in a try-except, so if anything goes wrong with trying to colorize the image, the flask app will return an error with the error traceback as the text content. If the frontend receives that kind of response, it will log the traceback to the console, display an error message on the frontend, and allow the user to upload a new image. We have not been successful in producing an error when using the app as inteneded, and any error produced display mesages that should explain to the user where they went wrong.

For static testing we determined we needed to use a linter that does both stylistic and logical linting. We found that flake8 is a strong favorite for performing this linting. We ran flake 8 on both of the python files in our app (app.py and image_colorizer.py). We found dozens of stylistic issues, ranging from incorrect indentation, to lines that were too long, to missing newlines, and changed all of them to match flake8 style. We also found and removed one unused import.

8 Limitations and Future Work

Our code follows good design principles, as most functionality is separated into various components. One improvement we could make would be separating the frontend javascript code more by functionality. However, the vast majority of the code is dedicated to image colorization, while one function is dedicated to handling tabs, so it doesn't entirely seem necessary to separate these two out. Separating the frontend and the backend entirely makes it very easy to swap out frontends and backends going forwards: we could develop various frontend interfaces and hook them up to the same backend, or create an entirely new backend and as long as it receives and returns an image, we could use the same frontend. One limitation of our code is that we do not have automate tests to perform on our platform. While we have a set of tests to perform by hand, creating some automated tests could serve as sanity checks so we make sure we don't break something when making later improvements. Another limitation is that our backend server, PythonAnywhere, does not allow for direct deployment. Instead, we have to copy and paste our code in to the platform, which opens up a huge opportunity for mistakes to be made. To mitigate this as much as possible, we created a task on the platform that runs nightly and pulls the latest code from github, but we still have to refresh the app by hand when changes to the backend are made. This did successfully prevent the issue of human error when copying and pasting.

Moving forward there are several improvements we would hope to make to our web app. One use we hoped to add was the ability to colorize black and white videos. We were able to find several resources to help create the code for the video colorizer, and we were somewhat successful in colorizing videos, however several major problems came up. When we provided the network with an input video recorded through the webcam, the kernel kept crashing. In addition, the video colorizer was extremely slow, which we knew would only slow down our app, and make it more difficult to use. Moving forward we would hope to be able to solve these issues by spending more time debugging the video colorizer code and working to speed up the process. We would also probably need to acquire more memory on the backend, and potentially queue requests and return them when completed via email. This would not require to much changing on the frontend: we would just have to add an email input and allow it to display a message upon clicking colorize rather than the resulting image. On the backend, it would require a pretty massive overhaul, as we would have to figure out how to send emails from python, how to queue tasks, and how to handle memory in such a way that we don't slow down or crash the app handling videos. We also would like to display some kind of confidence score on

the frontend in the future based on the colorized image. We have the placeholder code on the frontend, but have not determined how to do so on the backend. The change should be trivial once we determine how to get the score.

9 Bibliography

1. Cheng, Z., Yang, Q., Sheng, B.: Deep colorization. In: *Proceedings of the IEEE International Conference on Computer Vision*. (2015) 415–423

2. Zhang, R., Isola, P., Efros, A.: Colorful Image Colorization. In: *European Conference on Computer Vision*. (2016) 649–666.