CONTENTS

List of Figures

ABSTRACT

The objective of this project is to convert black and white image to colour image without human intervention. This is done through a deep learning concept called Convolutional Neural Network (CNN). The conventional way to do this is through the use of photo editors like photoshop which takes a lot of time.

In the model development, initially color images are converted to "LAB" color space. Then we will train our Convolutional Neural Network model with L channel (Black & white image) as input with its respective A and B color channels and the trained weights will be saved. Our model will then be able to predict the A and B channels for the black and white images it has never seen before. We plan to do this by combining the predicted "A" and "B" from the neural network with "L" values of the black and white image and then converting it to the RGB colorspace and save the output as an image.

CHAPTER-1

1. INTRODUCTION

1.1 Purpose of the Project

The main purpose in our project is to let machine produce colors for a grayscale image. Consider the grayscale photographs At first glance, hallucinating their colors seems daunting, since so much of the information (two out of the three dimensions) has been lost. Looking more closely, however, one notices that in many cases, the semantics of the scene and its surface texture provide ample cues for many regions in each image: the grass is typically green, the sky is typically blue, and the ladybug is most definitely red. Of course, these kinds of semantic priors do not work for everything, e.g., the croquet balls on the grass might not, in reality, be red, yellow, and purple though it's a pretty good guess.

However, for this project, our goal is not necessarily to recover the actual ground truth color, but rather to produce a plausible colorization that could potentially fool a human observer. Therefore, our task becomes much more achievable to model enough of the statistical dependencies between the semantics and the textures of grayscale images and their color versions in order to produce visually compelling results.

Fig. 1.1 Objective of the project

The above figure Fig. 1.1 has example input grayscale photos and output colorizations desired to be obtained by our algorithm. These examples are cases where our model should work well.

1.2 Problem with the Existing System

Today all the colorization process is done by hand in Photoshop. In short, a picture can take up to one month to colorize. It requires extensive research. A face alone needs up to 20 layers of pink, green and blue shades to get it just right.

When it comes to colorizing an image via machine learning concepts, others have noted the easy availability of training data, and previous works have trained convolutional neural networks (CNNs) to predict color on large datasets . However, the results from these previous attempts tend to look desaturated. One explanation is that uses loss functions that encourage conservative predictions. These losses are inherited from standard regression problems, where the goal is to minimize Euclidean error between an estimate and the ground truth. We instead utilize a loss tailored to the colorization problem.

As pointed out by previous attempts, color prediction is inherently multimodal, meaning, many objects can take on several plausible colorizations. For example, an apple is typically red, green, or yellow, but unlikely to be blue or orange.

1.3 Proposed System

We train a CNN to map from a grayscale input to a distribution over quantized color value outputs using the architecture shown in Figure 1.2 below. Architectural details are described in the supplementary materials on our project webpage1 , and the model is publicly available. In the following, we focus on the design of the objective function, and our technique for inferring point estimates of color from the predicted color distribution.

To appropriately model the multimodal nature of the problem, we predict a distribution of possible colors for each pixel. Furthermore, we re-weight the loss at training time to emphasize rare colors. This encourages our model to exploit the full diversity of the large-scale data on which it is trained. Lastly, we produce a final colorization by taking the annealed mean of the distribution.

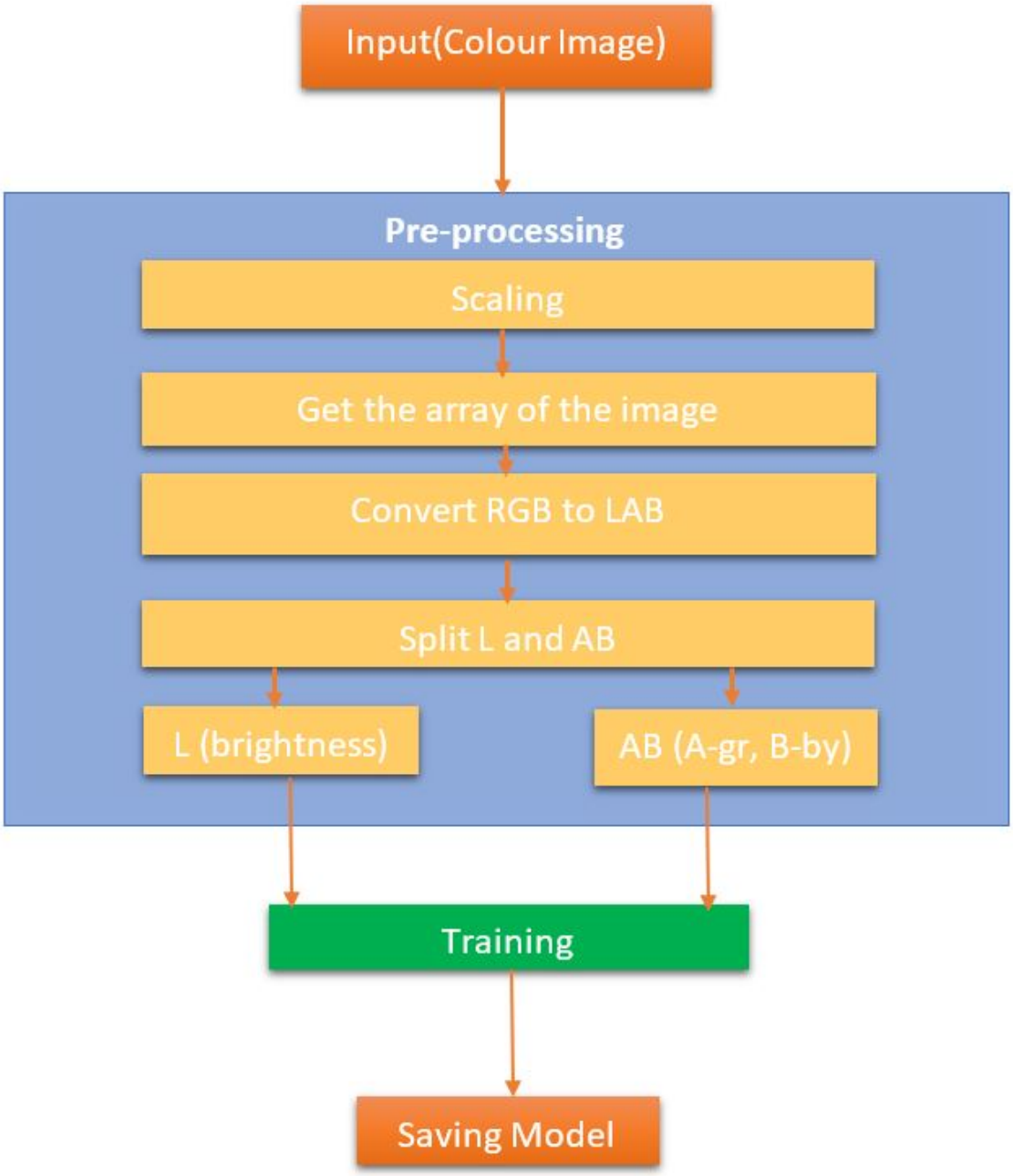


Fig. 1.2 Architecture for Training

The end result is colorizations that are more vibrant and perceptually realistic than those of previous approaches. Evaluating synthesized images is notoriously difficult . Since our ultimate goal is to make results that are compelling to a human observer, we introduce a novel way of evaluating colorization results, directly testing their perceptual realism.

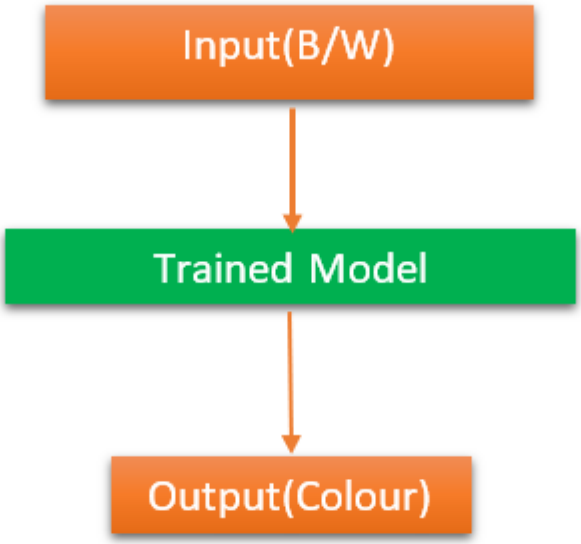


Fig 1.3 Architecture for Testing

We set up a "colorization Turing test," in which we show participants real and synthesized colors for an image, and ask them to identify the fake. In this quite difficult paradigm, we are able to fool participants on 32% of the instances (ground truth colorizations would achieve 50% on this metric), significantly higher than prior work .

This test demonstrates that in many cases, Colorful Image Colorization 3 our algorithm is producing nearly photorealistic results (see Figure 1 for selected examples). We also show that our system's colorizations are realistic enough to be useful for downstream tasks, in particular object classification, using an off-the-shelf VGG network. We additionally explore colorization as a form of self-supervised representation learning, where raw data is used as its own source of supervision.

The idea of learning feature representations in this way goes back at least to autoencoders . More recent works have explored feature learning via data imputation, where a held-out subset of the complete data is predicted. Our method follows in this line, and can be termed a cross-channel encoder. We test how well our model performs in generalization tasks,

compared to previous and concurrent self-supervision algorithms, and find that our method performs surprisingly well, achieving state-of-the-art performance on several metrics.

Our contributions in this paper are in two areas. First, we make progress on the graphics problem of automatic image colorization by

- Designing an appropriate objective function that handles the multimodal uncertainty of the colorization problem and captures a wide diversity of colors.
- Introducing a novel framework for testing colorization algorithms, potentially applicable to other image synthesis tasks, and
- Setting a new high-water mark on the task by training on a million color photos. Secondly, we introduce the colorization task as a competitive and straightforward method for self-supervised representation learning, achieving state-of-the-art results on several benchmarks.

Colorization algorithms mostly differ in the ways they obtain and treat the data for modeling the correspondence between grayscale and color. Non-parametric methods, given an input grayscale image, first define one or more color reference images (provided by a user or retrieved automatically) to be used as source data. Then, following the Image Analogies framework , color is transferred onto the input image from analogous regions of the reference images.

Parametric methods, on the other hand, learn prediction functions from large datasets of color images at training time, posing the problem as either regression onto continuous color space or classification of quantized color values.

Our method also learns to classify colors, but does so with a larger model, trained on more data, and with several innovations in the loss function and mapping to a final continuous output.

1.3.1 Technologies Used

The following are the base technologies used to develop the application.

1.3.1.1 Python

Python is an interpreted high-level programming language for general-purpose programming. Created by Guido van Rossum and first released in 1991, Python has a design philosophy that emphasizes code readability, notably using significant whitespace. It provides constructs that enable clear programming on both small and large scales.

Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms, including object-oriented, imperative, functional and procedural, and has a large and comprehensive standard library.

Most machine learning algorithms and applications are based on Python programming language due to its high ease of implementation, readability and writability. So we will be using most of the packages specially designed for machine learning in Python.

1.3.1.2 Convolutional Neural Networks

Convolutional Neural Networks are very similar to ordinary Neural Networks. they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer and all the tips/tricks we developed for learning regular Neural Networks still apply.

So what does change? ConvNet architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the amount of parameters in the network.

Convolutional Neural Networks is a type of advanced artificial neural network. It differs from regular neural networks in terms of the flow of signals between neurons. Typical neural networks pass signals along the input-output channel in a single direction, without allowing signals to loop back into the network. This is called a forward feed.

While forward feed networks were successfully employed for image and text recognition, it required all neurons to be connected, resulting in an overly-complex network structure. The cost of complexity grows when the network has to be trained on large datasets which, coupled with the limitations of computer processing speeds, result in grossly long training times. Hence, forward feed networks have fallen into disuse from mainstream machine learning in today's high resolution, high bandwidth, mass media age. A new solution was needed.

In 1986, researchers Hubel and Wiesel were examining a cat's visual cortex when they discovered that its receptive field comprised sub-regions which were layered over each other to cover the entire visual field. These layers act as filters that process input images, which are then passed on to subsequent layers. This proved to be a simpler and more efficient way to carry signals. In 1998, Yann LeCun and Yoshua Bengio tried to capture the organization of neurons in the cat's visual cortex as a form of artificial neural net, establishing the basis of the first CNN.

Parts in CNN:

There are four main parts in CNN: convolution, subsampling, activation and full connectedness.

1. Convolution

The first layers that receive an input signal are called convolution filters. Convolution is a process where the network tries to label the input signal by referring to what it has learned in the past. If the input signal looks like previous cat images it has seen before, the "cat" reference signal will be mixed into, or convolved with, the input signal. The resulting output signal is then passed on to the next layer.

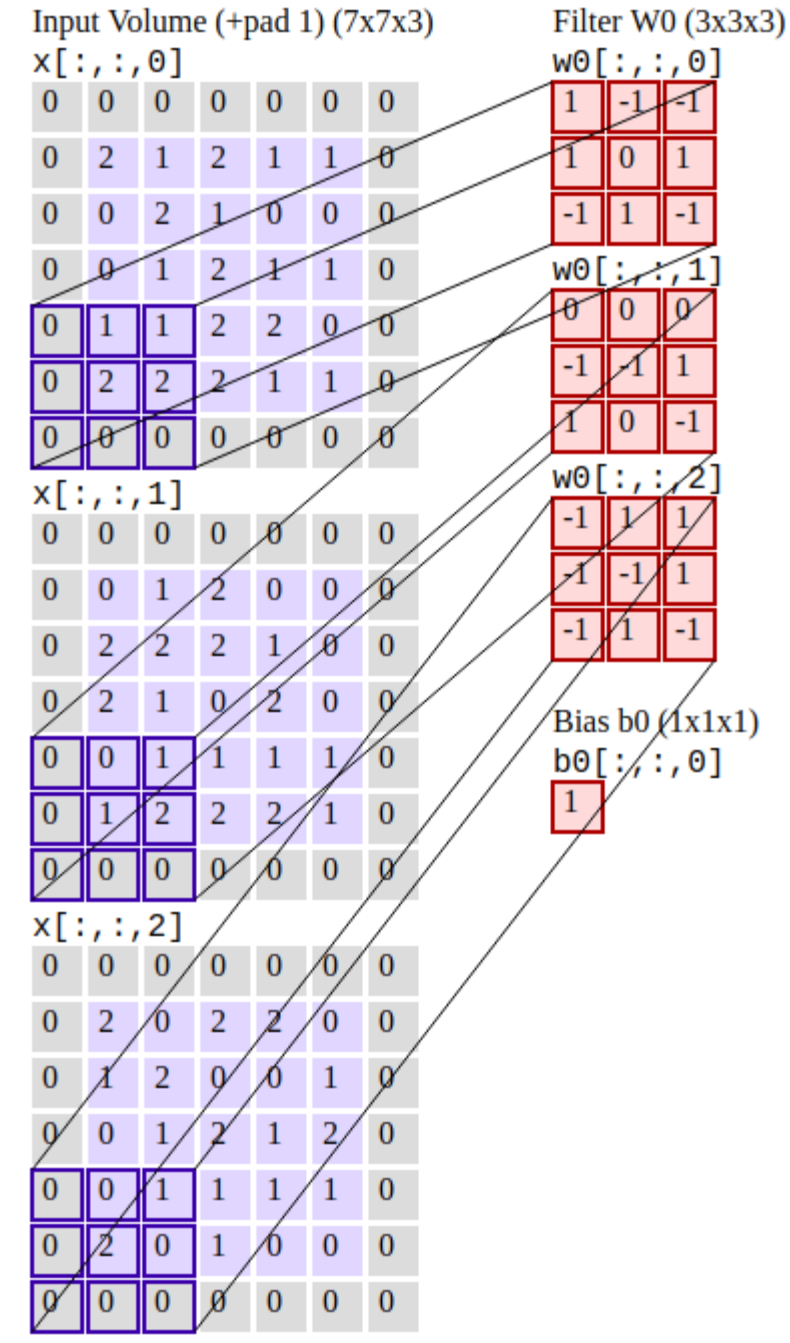


Fig 1.4 Convolution example 1

Each convolution filter represents a feature of interest (e.g whiskers, fur), and the CNN algorithm learns which features comprise the resulting reference (i.e. cat). The output signal strength is not dependent on where the features are located, but simply whether the features

are present. Hence, a cat could be sitting in different positions, and the CNN algorithm would still be able to recognize it.

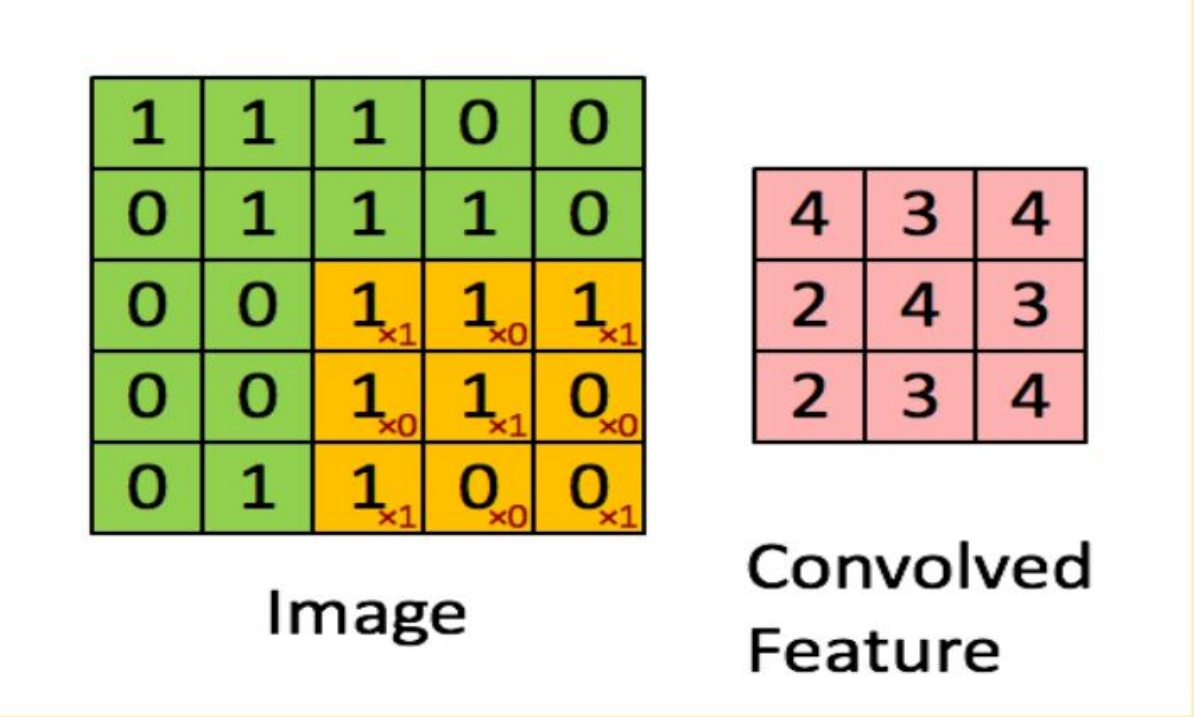


Fig 1.5 Convolution example 2

The "window" that moves over the image is called a kernel. Kernels are typically square and 3x3 is a fairly common kernel size for small-ish images. The distance the window moves each time is called the stride. Additionally of note, images are sometimes padded with zeros around the perimeter when performing convolutions, which dampens the value of the convolutions around the edges of the image (the idea being typically the center of photos matter more).

2. Spatial Pooling

Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information. Spatial Pooling can be of different types: Max, Average, Sum etc.

In case of Max Pooling, we define a spatial neighborhood (for example, a 2×2 window) and take the largest element from the rectified feature map within that window. Instead of taking

the largest element we could also take the average (Average Pooling) or sum of all elements in that window.

Taking the maximum is called Max Pooling.

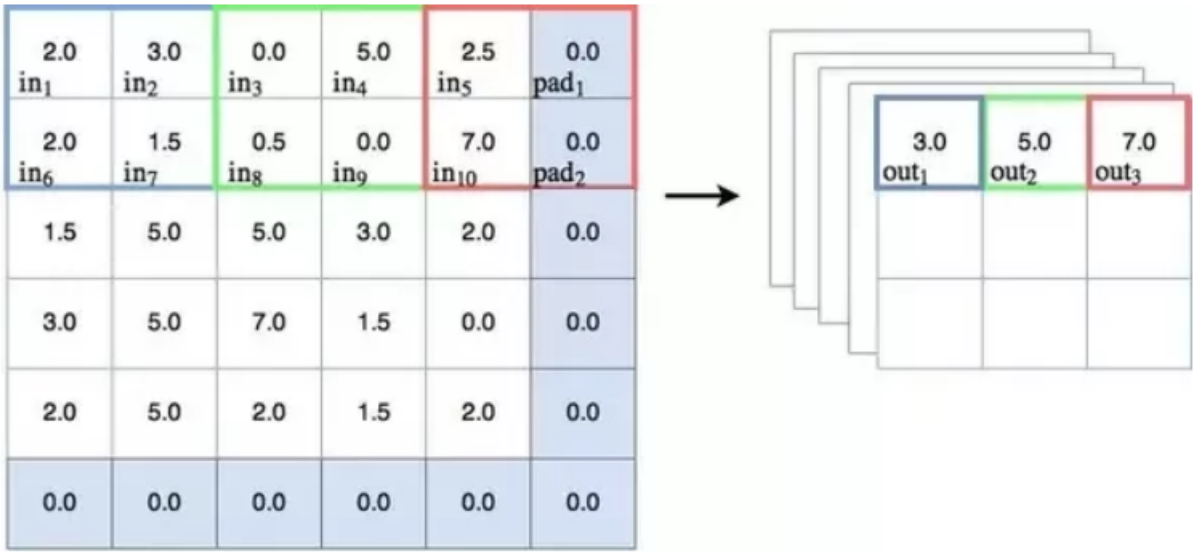


Fig 1.6 Max pooling

Taking the average is called Average Pooling.

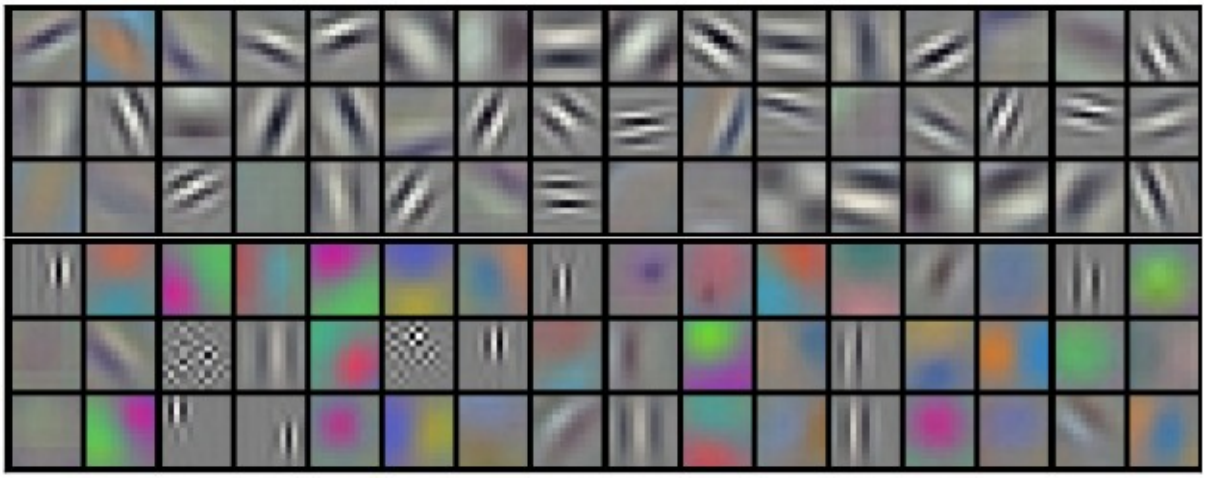


Fig 1.7 Feature detection visualization

Examples of subsampling methods (for image signals) include reducing the size of the image, or reducing the color contrast across red, green, blue (RGB) channels.

3. Activation Layer

The activation layer controls how the signal flows from one layer to the next, emulating how neurons are fired in our brain. Output signals which are strongly associated with past references would activate more neurons, enabling signals to be propagated more efficiently for identification.

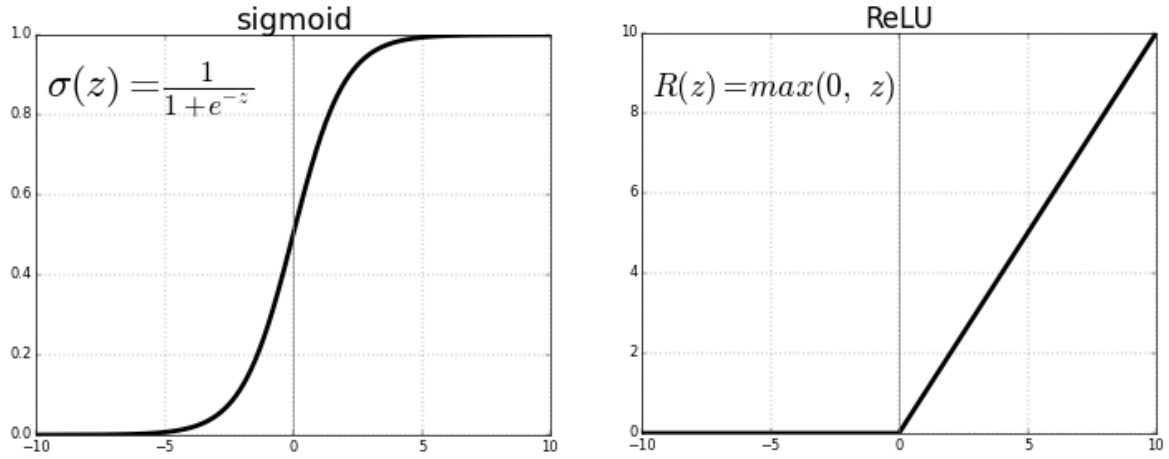


Fig 1.8. Sigmoid and Relu Activation

ReLU is an element wise operation (applied per pixel) and replaces all negative pixel values in the feature map by zero. The purpose of ReLU is to introduce non-linearity in our ConvNet, since most of the real-world data we would want our ConvNet to learn would be non-linear (Convolution is a linear operation – element wise matrix multiplication and addition, so we account for non-linearity by introducing a nonlinear function like ReLU).

The ReLU operation can be understood clearly from figure below. It shows the ReLU operation applied to one of the feature maps obtained in figure above. The output feature map here is also referred to as the 'Rectified' feature map.

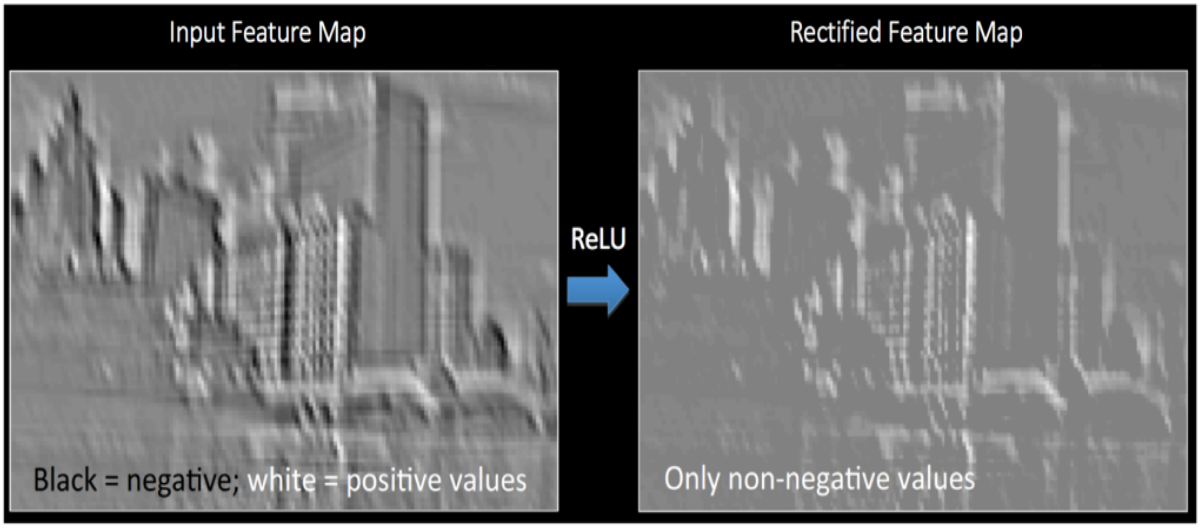


Fig 1.9 Importance of using Relu in CNN

CNN is compatible with a wide variety of complex activation functions to model signal propagation, the most common function being the Rectified Linear Unit (ReLU), which is favoured for its faster training speed.

4. Fully Connected Layer

The Fully Connected layer is a traditional Multi Layer Perceptron that uses a softmax activation function in the output layer (other classifiers like SVM can also be used, but will stick to softmax in this post). The term “Fully Connected” implies that every neuron in the previous layer is connected to every neuron on the next layer. I recommend reading this post if you are unfamiliar with Multi Layer Perceptrons.

The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the Fully Connected layer is to use these features for classifying the input image into various classes based on the training dataset. For example, the image classification task we set out to perform has fohttp://cs231n.github.io/neural-networks-1/ur possible outputs as shown in Figure 14 below (note that Figure 14 does not show connections between the nodes in the fully connected layer)

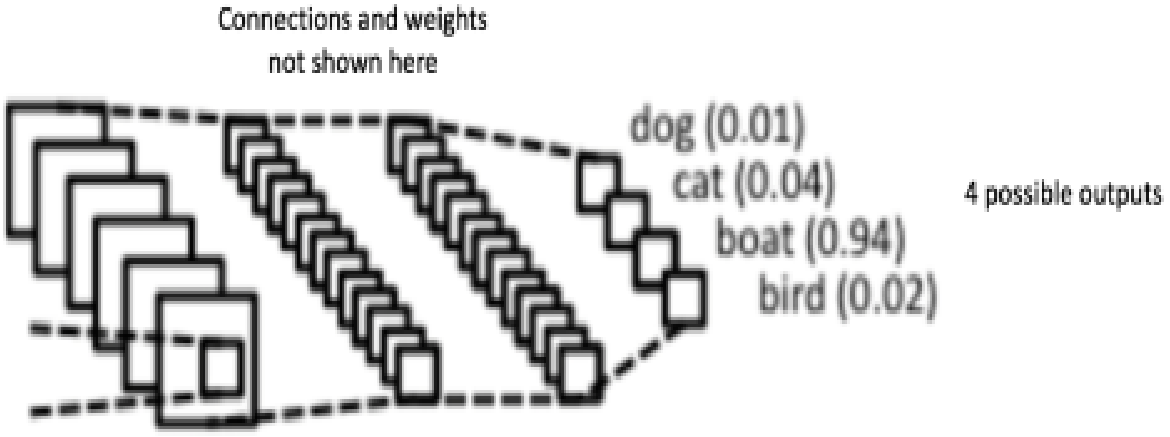


Fig 1.10 Fully Connected Layer -each node is connected to every other node in the adjacent layer

Apart from classification, adding a fully-connected layer is also a (usually) cheap way of learning non-linear combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better .

The sum of output probabilities from the Fully Connected Layer is 1. This is ensured by using the Softmax as the activation function in the output layer of the Fully Connected Layer. The Softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one.

5.Putting it all together-Training using Backpropagation

As discussed above, the Convolution + Pooling layers act as Feature Extractors from the input image while Fully Connected layer acts as a classifier.

Note that in Figure 15 below, since the input image is a boat, the target probability is 1 for Boat class and 0 for other three classes, i.e.

- Input Image = Boat
- Target Vector = [0, 0, 1, 0]

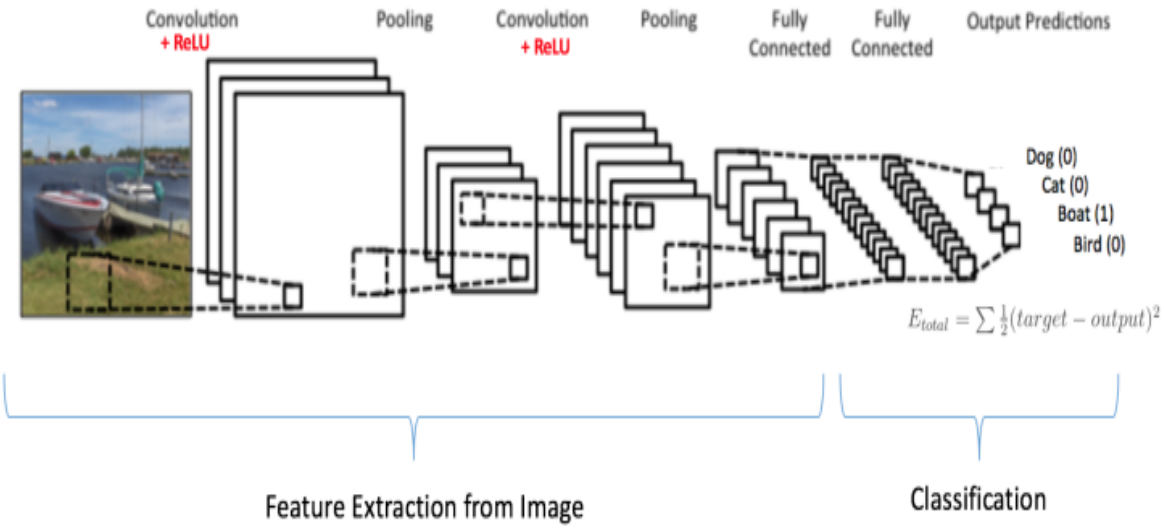


Fig 1.11 Training the ConvNet

The overall training process of the Convolution Network may be summarized as below:

- Step1: We initialize all filters and parameters / weights with random values
- Step2: The network takes a training image as input, goes through the forward propagation step (convolution, ReLU and pooling operations along with forward propagation in the Fully Connected layer) and finds the output probabilities for each class.
    - Let's say the output probabilities for the boat image above are [0.2, 0.4, 0.1, 0.3]
    - Since weights are randomly assigned for the first training example, output probabilities are also random.
- Step3: Calculate the total error at the output layer (summation over all 4 classes)
    - Total Error = ∑ ½ (target probability – output probability) ²
- Step4: Use Backpropagation to calculate the gradients of the error with respect to all weights in the network and use gradient descent to update all filter values / weights and parameter values to minimize the output error.

- The weights are adjusted in proportion to their contribution to the total error.
- When the same image is input again, output probabilities might now be [0.1, 0.1, 0.7, 0.1], which is closer to the target vector [0, 0, 1, 0].
- This means that the network has learnt to classify this particular image correctly by adjusting its weights / filters such that the output error is reduced.
- Parameters like number of filters, filter sizes, architecture of the network etc. have all been fixed before Step 1 and do not change during training process – only the values of the filter matrix and connection weights get updated.

- Step5: Repeat steps 2-4 with all images in the training set.

The above steps train the ConvNet – this essentially means that all the weights and parameters of the ConvNet have now been optimized to correctly classify images from the training set.

When a new (unseen) image is input into the ConvNet, the network would go through the forward propagation step and output a probability for each class (for a new image, the output probabilities are calculated using the weights which have been optimized to correctly classify all the previous training examples). If our training set is large enough, the network will (hopefully) generalize well to new images and classify them into correct categories.

1.3.1.3 Django Web Framework

Django is a free and open-source web framework, written in Python, which follows the model-view-template (MVC) architectural pattern. It is maintained by the Django Software Foundation (DSF), an independent organization established as a 501(c)(3) non-profit.

Django's primary goal is to ease the creation of complex, database-driven websites. Django emphasizes reusability and "pluggability" of components, less code, low coupling, rapid development, and the principle of don't repeat yourself. Python is used throughout, even for settings files and data models. Django also provides an optional administrative create, read, update and delete interface that is generated dynamically through introspection and configured via admin models.

We use Django Web Framework as a front-end tool to develop a web application that leverages the functionality of the underlying machine learning application.

1.3.1.4 Google Colaboratory

Colab is a Google's collaborative version of the Jupyter/iPython notebook.Now you can use Nvidia Tesla K80 GPU for free. You do not have to pay Amazon Web Services (AWS) anymore! Google released it to the public with the aim of improving the machine learning education and research. We can develop deep learning applications with Google Colaboratory -on the free Tesla K80 GPU- using Keras, Tensorflow and PyTorch.

If you click the Colab website, the following screen will welcome you.

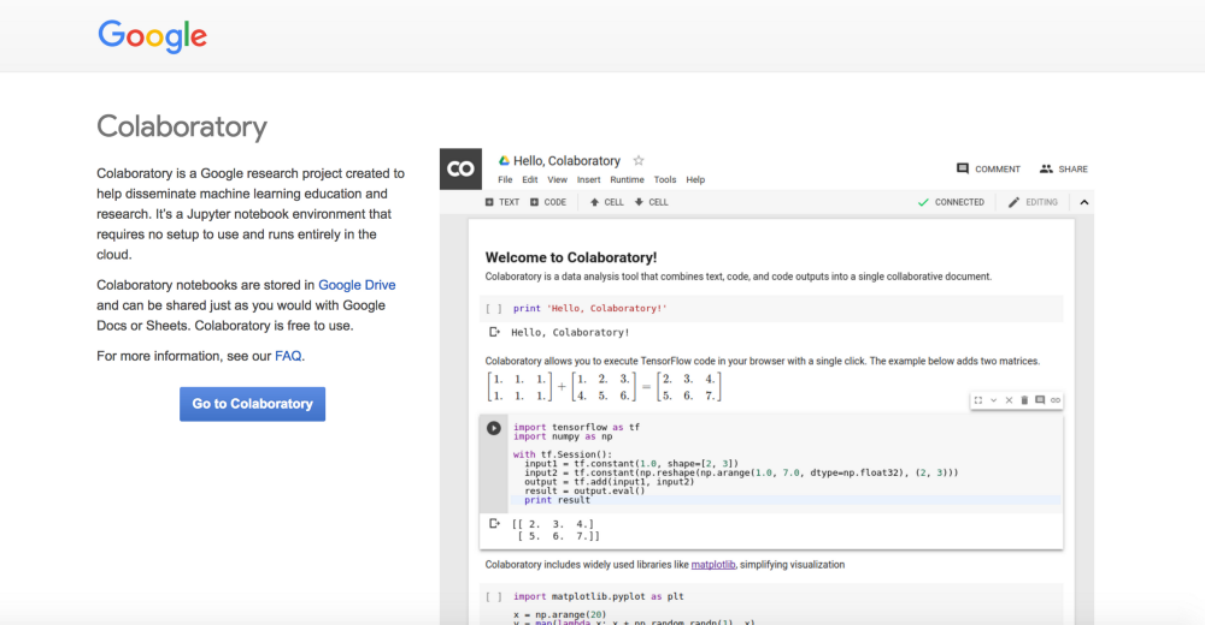


Fig 1.12 Google Colab welcome screen.

Click the "Go to Colaboratory" button and sign in with your Google account. And then, open a New Python 3 notebook as you can see below.

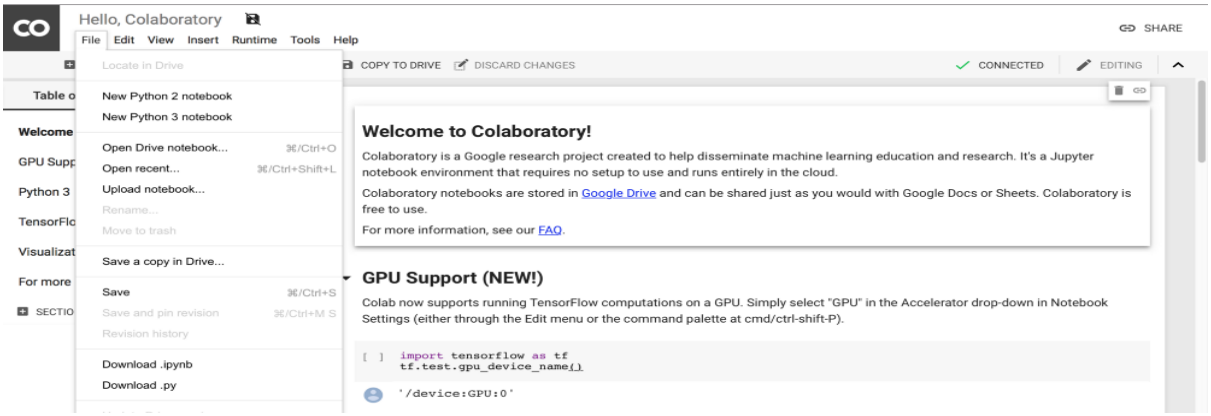


Fig 1.13 Opening new ipython notebook

After this step, you should see that a Jupyter notebook was opened. You are seeing now one empty cell with “Play” button. After the code is written, you can run the cell using the play button.

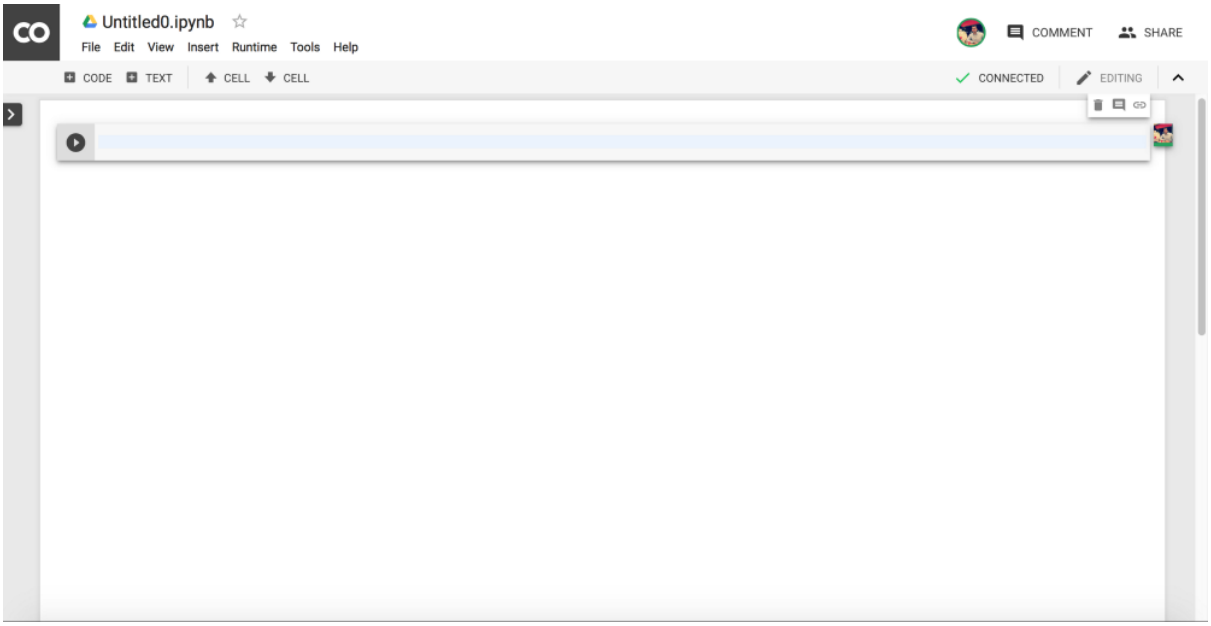


Fig 1.14 New iPython Notebook

To further use it we need to install ocaml fuse to access the drive as your local file system.

Setting up Free GPU in colab is so simple. just follow Edit > Notebook settings or Runtime>Change runtime type and select GPU as Hardware accelerator.

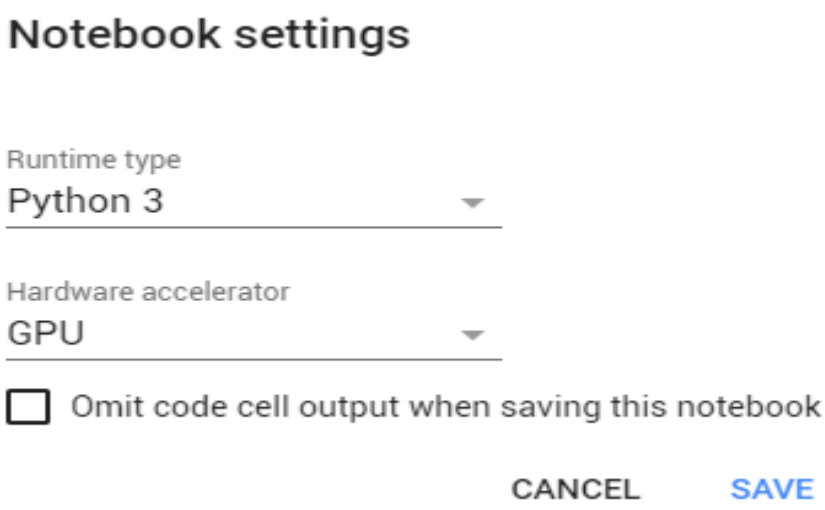


Fig 1.15 Setting up free GPU

1.3.2 Approach

Our project processes the black and white image as follows - this is a basic outline on how to render an image, the basics of digital colors, and the main logic for our neural network.

Black and white images can be represented in grids of pixels. Each pixel has a value that corresponds to its brightness. The values span from 0 - 255, from black to white.

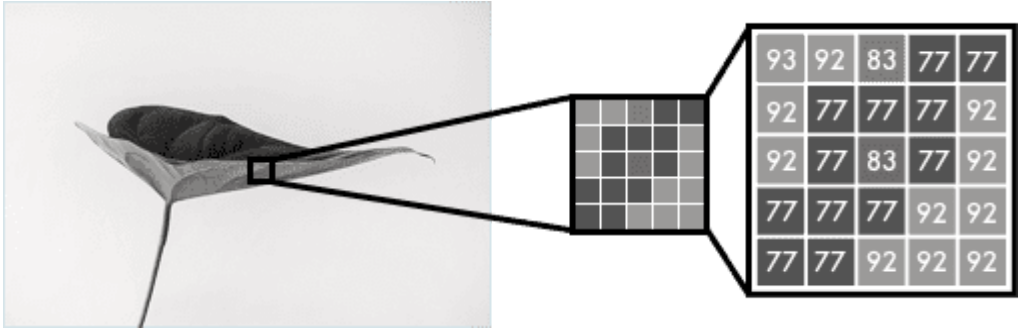


Fig 1.16 Rendering a B&W image

Color images consist of three layers: a red layer, a green layer, and a blue layer. This might be counter-intuitive to you. Imagine splitting a green leaf on a white background into the three channels. Intuitively, you might think that the plant is only present in the green layer.

But, as you see below, the leaf is present in all three channels. The layers not only determine color, but also brightness.

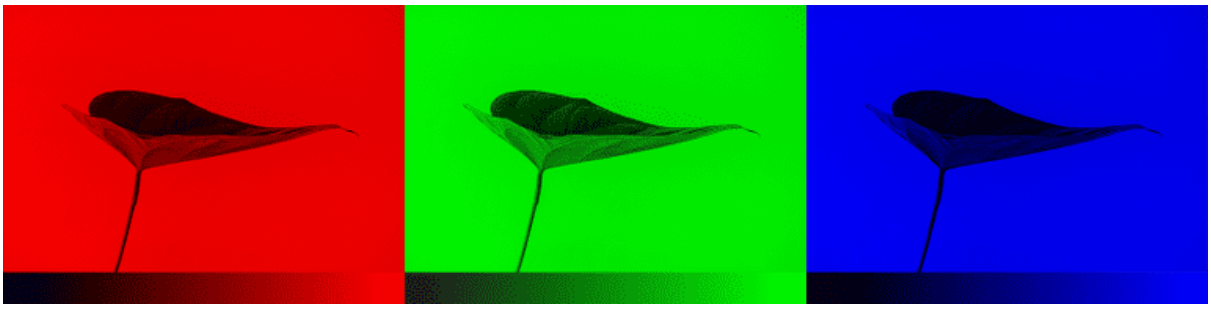


Fig 1.17 RGB Explanation 1

To achieve the color white, for example, you need an equal distribution of all colors. By adding an equal amount of red and blue, it makes the green brighter. Thus, a color image encodes the color and the contrast using three layers:

Fig 1.18 RGB Explanation 2

Just like black and white images, each layer in a color image has a value from 0 - 255. The value 0 means that it has no color in this layer. If the value is 0 for all color channels, then the image pixel is black.

As you may know, a neural network creates a relationship between an input value and output value. To be more precise with our colorization task, the network needs to find the traits that link grayscale images with colored ones.

In sum, we are searching for the features that link a grid of grayscale values to the three color grids.

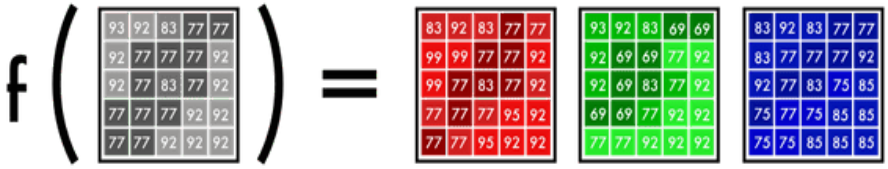


Fig 1.19 RGB Explanation 3

The first section breaks down the core logic. We'll build a bare-bones 40-line neural network as an "Alpha" colorization bot. There's not a lot of magic in this code snippet - which is helpful so that we can get familiar with the syntax. The next step is to create a neural network that can generalize - our final version. We'll be able to color images the bot has not seen before.

We'll start by making a simple version of our neural network to color an image of a woman's face. This way, you can get familiar with the core syntax of our model as we add features to it.

With just 40 lines of code, we can make the following transition. The middle picture is done with our neural network and the picture to the right is the original color photo. The network is trained and tested on the same image.

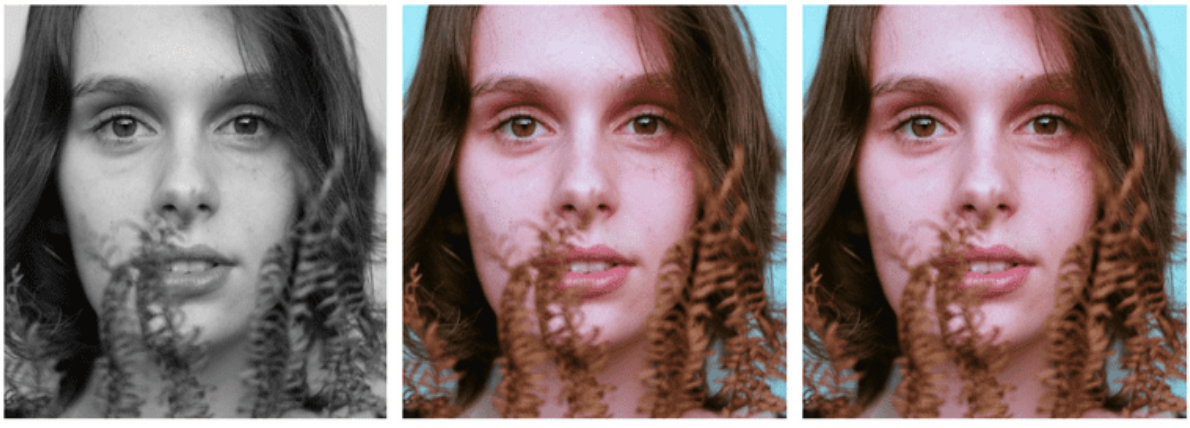


Fig 1.20 Single image trained result

1.3.2.1 Color space

First, we'll use an algorithm to change the color channels, from RGB to Lab. L stands for lightness, and a and b for the color spectrums green–red and blue–yellow.

As you can see below, a Lab encoded image has one layer for grayscale and have packed three color layers into two. This means that we can use the original grayscale image in our final prediction. Also, we only have to two channels to predict.

Fig 1.21 "Lab" Demonstration

Science fact - 94% of the cells in our eyes determine brightness. That leaves only 6% of our receptors to act as sensors for colors. As you can see in the above image, the grayscale image is a lot sharper than the color layers. This is another reason to keep the grayscale image in our final prediction.

1.3.2.2 From B&W to color

Our final prediction looks like this. We have a grayscale layer for input, and we want to predict two color layers, the ab in Lab. To create the final color image we'll include the L/grayscale image we used for the input, thus, creating a Lab image.

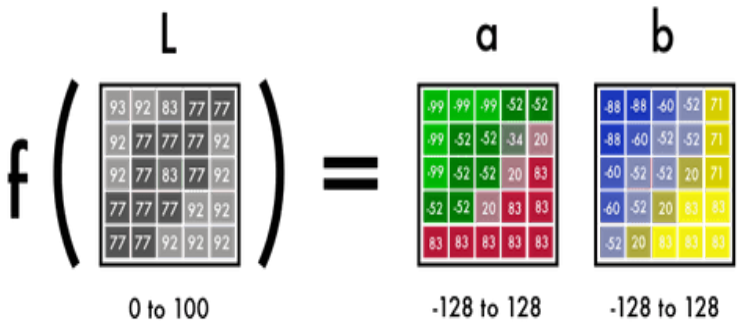


Fig 1.22 "Lab" Explanation

To turn one layer into two layers, we use convolutional filters. Think of them as the blue/red filters in 3D glasses. Each filter determines what we see in a picture. They can highlight or remove something to extract information out of the picture. The network can either create a new image from a filter or combine several filters into one image.

For a convolutional neural network, each filter is automatically adjusted to help with the intended outcome. We'll start by stacking hundreds of filters and narrow them down into two layers, the a and b layers.

To understand the weakness of the Alpha-version, try coloring an image it has not been trained on. If you try it, you'll see that it makes a poor attempt. It's because the network has memorized the information. It has not learned how to color an image it hasn't seen before. But this is what we'll do in the Beta-version - we'll teach our network to generalize.

1.3.2.3 The feature extractor

Our neural network finds characteristics that link grayscale images with their colored versions.

Imagine you had to color black and white images - but with restriction that you can only see nine pixels at a time. You could scan each image from the top left to bottom right and try to predict which color each pixel should be.

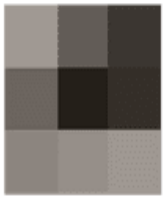


Fig 1.23 Basic Feature Example

For example, these nine pixels is the edge of the nostril from the woman just above. As you can imagine, it'd be next to impossible to make a good colorization, so you break it down into steps.

First, you look for simple patterns: a diagonal line, all black pixels, and so on. You look for the same exact pattern in each square and remove the pixels that don't match. You generate 64 new images from your 64 mini filters.

If you scan the images again, you'd see the same small patterns you've already detected. To gain a higher level understanding of the image, you decrease the image size in half.

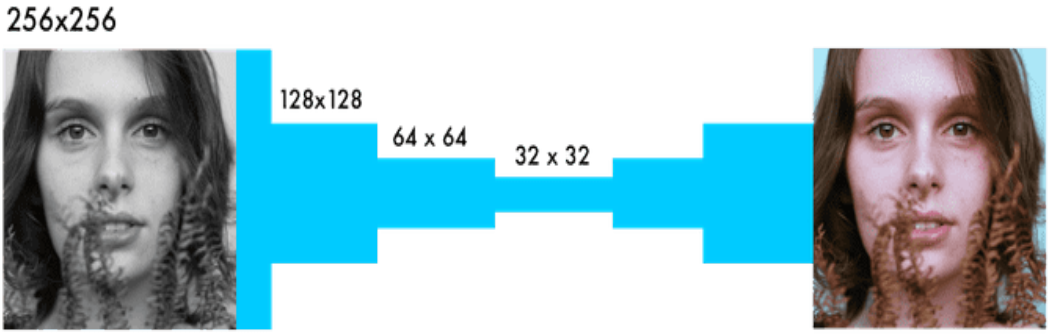


Fig 1.24 Feature extraction explanation

You still only have a three by three filter to scan each image. But by combining your new nine pixels with your lower level filters you can detect more complex patterns. One pixel combination might form a half circle, a small dot, or a line. Again, you repeatedly extract the same pattern from the image. This time, you generate 128 new filtered images.

As mentioned, you start with low-level features, such as an edge. Layers closer to the output are combined into patterns, then into details, and eventually transformed into a face. The process is like most neural networks that deal with vision, known as convolutional neural

networks. Convolution is similar to the word combine, you combine several filtered images to understand the context in the image.

1.3.2.4 From feature extraction to color

The neural network operates in a trial and error manner. It first makes a random prediction for each pixel. Based on the error for each pixel, it works backward through the network to improve the feature extraction.

It starts adjusting for the situations that generate the largest errors. In this case, it's whether to color or not and to locate different objects. Then it colors all the objects brown. It's the color that is most similar to all other colors, thus producing the smallest error.

Because most of the training data is quite similar, the network struggles to differentiate between different objects. It will adjust different tones of brown, but may fail to generate more nuanced colors.

The main difference from other visual networks is the importance of pixel location. In coloring networks, the image size or ratio stays the same throughout the network. In other networks, the image gets distorted the closer it gets to the final layer.

The max-pooling layers in classification networks increase the information density, but also distort the image. It only values the information, but not the layout of an image. In coloring networks we instead use a stride of 2, to decrease the width and height by half. This also increases information density but does not distort the image.

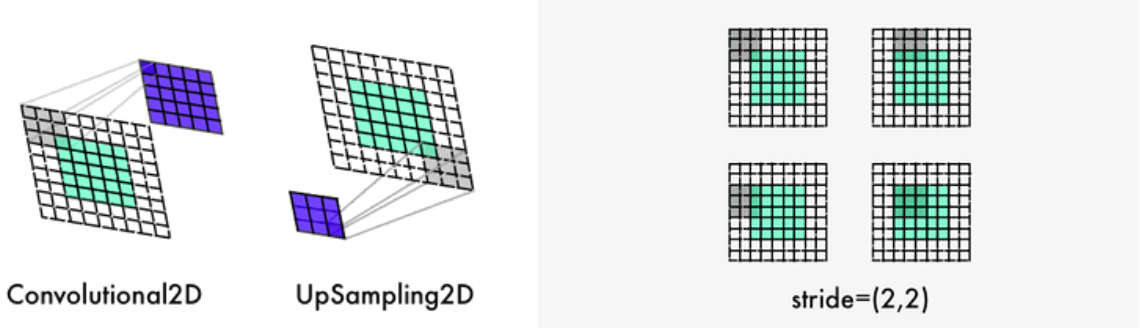


Fig 1.25 CNN Layer Types

Two further differences are upsampling layers and maintaining the image ratio. Classification networks only care about the final classification. Therefore, they keep decreasing the image size and quality as it moves through the network.

Coloring networks keep the image ratio. This is done by adding white padding like the visualization above. Otherwise, each convolutional layer cuts the images.

To double the size of the image, the coloring network uses an upsampling layer.

1.4 Scope of the Project

The scope of our product is to provide an efficient and enhanced software tool for the users to perform colorization of black and white images in research, academic, governmental and business organizations that are having large pool of imaged, scanned images with respect to the size of images and the type of grayscale images, the product is recognizing them, searching them and processing them faster according to the needs of the environment. Our project does not include rectifying classification problem to colorizing images uses it is a pure CNN based model.

CHAPTER-2

2.SOFTWARE REQUIREMENTS SPECIFICATION

2.1 Requirements Specification Document

A software requirements specification (SRS) is a description of a software system to be developed. It lays out functional and non-functional requirements, and may include a set of use cases that describe user interactions that the software must provide.

Software requirements specification establishes the basis for an agreement between customers and contractors or suppliers (in market-driven project, these roles may be played by the marketing and development divisions) on what the software product is to do as well as what it is not expected to do. Software requirements specification permits a rigorous assessment of requirements before design can begin and reduces later redesign.

2.1.1 Functional Requirements

- Load the dataset of images with different resolutions
- Training the system with the colour images
- Processing the gray-scale images through the net.
- Encoding the image. (Segmentation)
- Extracting the Features from the image.
- Recognizing the colors in images.

2.1.2 Non Functional Requirements

- Performance, Response Time, Throughput
- Serviceability
- Data Integrity
- Usability & Interoperability

2.2 Software Requirements

- Operating System : Windows 7,8,10 / Linux/ Mac
- Platform : Anaconda (optional)
- Programming Language : Python
- Web Framework : Django

2.2.1 Python Packages

Python packages required to run this application

- Tensorflow
- Tensorflow-gpu
- CUDA 8.0
- Django
- Keras
- Skimage
- Numpy
- Open Cv
- Pil
- Resize image

2.3 Hardware Requirements

- Processor : Intel i5
- Hard Disk : 40GB HDD
- RAM : 8 GB
- GPU : GTX Nvidia 4 GB

CHAPTER-3

3. LITERATURE SURVEY

3.1 Colourization using Regression

Authors: Shiguang Liu, XiangZhang

Previous works have not used deep learning. They used regression to predict the colour of each pixel. This , however, produces fairly bland and dull results.

Fig 3.1 Results of a regression based model. Left: Input to the model. Right: Output of the model.

Previous works used Mean Squared Error (MSE) as the loss function to train the model. The authors noted that MSE will try to 'average' out the colors in order to get the least average error, which will result in a bland look. The authors instead pose the task of colorizing pictures as a classification problem.

3.2 Colorful Image Colorization ECCV 5th October 2015

Authors: Richard Zhang, Phillip Isola, Alexei A. Efros

In 2015, Richard Zhang, Phillip Isola and Alexei Efros have published a research paper on colorization of the black and white photos. In their paper, given a grayscale photograph as input, this paper attacks the problem of hallucinating a plausible color version of the photograph. This problem is clearly underconstrained, so previous approaches have either

relied on significant user interaction or resulted in desaturated colorizations. They proposed a fully automatic approach that produces vibrant and realistic colorizations. They embraced the underlying uncertainty of the problem by posing it as a classification task and use class-rebalancing at training time to increase the diversity of colors in the result.

The system is implemented as a feed-forward pass in a CNN at test time and is trained on over a million color images. They have evaluated their algorithm using a "colorization Turing test," asking human participants to choose between a generated and ground truth color image. Their method successfully fools humans on 32% of the trials, significantly higher than previous methods. Moreover, they have shown that colorization can be a powerful pretext task for self-supervised feature learning, acting as a cross-channel encoder. This approach results in state-of-the-art performance on several feature learning benchmarks.

We have taken the concepts of having only two channels, a and b in Lab format from this research paper. The fact that training the model to predict 3 channels of colors will take up a much larger training time and power utilization. Due to these reasons, the color channels a and b are used. Training only two channels of colors with inclusion of L will make the training much quicker. Not only did the training channels get reduced by one, but the intensity of the color is depended on the grayscale value from L channel, so only a value from -1 to 1 can produce the color equivalent of 2 color channels - green and pink; or blue and yellow.

3.3 Colorizing B&W Photos with Neural Networks October 13, 2017

Author: Emil Wallner

Emil Wallner, one of the famous open source software developer has worked on the problem of colorization in 2017 and posted the project in GitHub along with a well documented article in FloydHub.

In his article, he has mentioned the approach very similar to our design. He has divided the project into 3 level. They are as follows:

- **Alpha version:** Alpha version is the design where he had trained a Convolutional Neural Network with a single image. This model has become the proof of concept for his design.

- **Beta version:** Beta version is where the model from alpha version is trained with more than one image. He called this process generalization as the trained model will be tested for any generic black and white image.

- **Final version:** The final version of his design has improvised beta version where in the encoding layers are coupled with an individually trained image classifier. The results from the encoding layer and classifier is fused together which he called as Fusion Layer. The result from the fusion layer are then fed to decoding into the predicted ab color channel values which are used to produce a color image.

The author of the article had considered the colorization of black and white images as a classification problem. He had hence used a proven pre-trained classifier to the convolutional layers. Our architecture is proven to be better than the beta version of his work. This is possible even without the use of the classifier. Moreover, the classification of images took rather high response time than the model we have designed.

CHAPTER-4

4. SYSTEM DESIGN

4.1 Introduction to UML

The Unified Modeling Language (UML) is a standard visual modeling language intended to be used for modeling business and similar processes and analysis, design, and implementation of software-based systems

UML is a common language for business analysts, software architects and developers used to describe, specify, design, and document existing or new business processes, structure and behavior of artifacts of software systems.

UML can be applied to diverse application domains (e.g., banking, finance, internet, aerospace, healthcare, etc.) It can be used with all major object and component software development methods and for various implementation platforms (e.g., J2EE, .NET).

It is a standard modeling language, not a software development process. UML Specification explained that process:

- provides guidance as to the order of a team's activities,
- specifies what artifacts should be developed,
- directs the tasks of individual developers and the team as a whole, and
- offers criteria for monitoring and measuring a project's products and activities.

UML is intentionally process independent and could be applied in the context of different processes. Still, it is most suitable for use case driven, iterative and incremental development processes. An example of such process is Rational Unified Process (RUP).

UML is not complete and it is not completely visual. Given some UML diagram, we can't be sure to understand depicted part or behavior of the system from the diagram alone. Some information could be intentionally omitted from the diagram, some information represented on the diagram could have different interpretations, and some concepts of UML have no graphical notation at all, so there is no way to depict those on diagrams.

4.2 UML Diagrams

4.2.1 Use Case Diagram

A use case diagram at its simplest is a representation of a user's interaction with the system that shows the relationship between the user and the different use cases in which the user is involved. A use case diagram can identify the different types of users of a system and the different use cases and will often be accompanied by other types of diagrams as well.

While a use case itself might drill into a lot of detail about every possibility, a use-case diagram can help provide a higher-level view of the system. It has been said before that "Use case diagrams are the blueprints for your system". They provide the simplified and graphical representation of what the system must actually do.

Due to their simplistic nature, use case diagrams can be a good communication tool for stakeholders. The drawings attempt to mimic the real world and provide a view for the stakeholder to understand how the system is going to be designed. Siau and Lee conducted research to determine if there was a valid situation for use case diagrams at all or if they were unnecessary. What was found was that the use case diagrams conveyed the intent of the system in a more simplified manner to stakeholders and that they were "interpreted more completely than class diagrams".

The purpose of the use case diagrams is simply to provide the high level view of the system and convey the requirements in layman's terms for the stakeholders. Additional diagrams and documentation can be used to provide a complete functional and technical view of the system.

To model a system, the most important aspect is to capture the dynamic behaviour. To clarify bit in details, dynamic behaviour means the behaviour of the system when it is running /operating. The diagram is used to model the system/subsystem of an application.

So only static behaviour is not sufficient to model a system rather dynamic behaviour is more important than static behaviour. In UML there are five diagrams available to model dynamic nature and use case diagram is one of them. Now as we have to discuss that the use case

diagram is dynamic in nature there should be some internal or external factors for making the interaction.

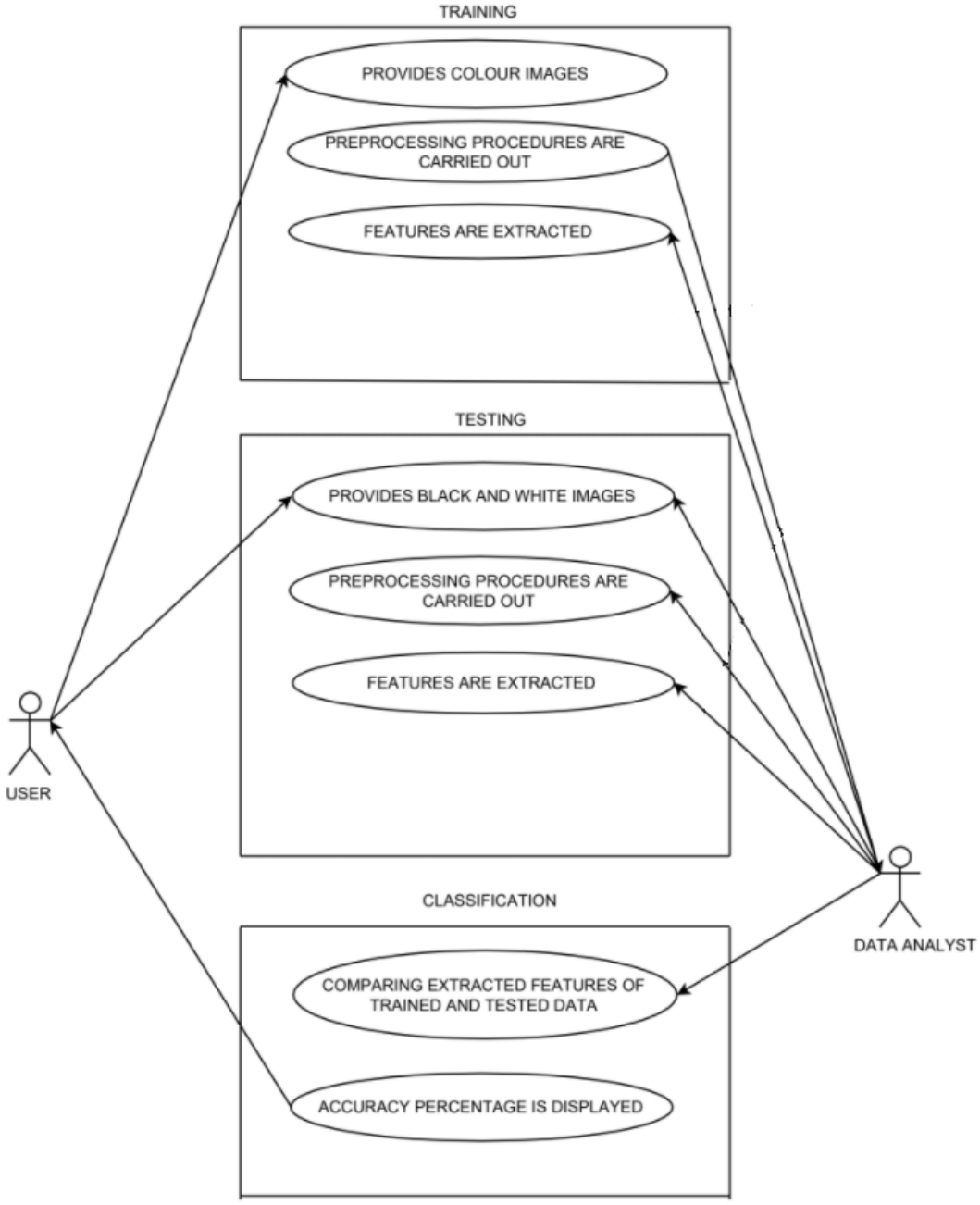
These internal and external agents are known as actors. So use case diagrams are consists of factors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system. So to model the entire system numbers of use case diagrams are used.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Now as we have to discuss that the use case diagram is dynamic in nature there should be some internal or external factors for making the interaction. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified.

In below we clearly explained about the use case functionality. In this we provided a description about :-

- Use case name : Data analyst
- Details: Data analyst is the only person who processes the images. He collects the images i.e dataset and preprocesses them and trains the collection and recognises the colors from the image
- Actors:
    - Primary actor: data analyst
    - Secondary actor: user

In the below stated the use case diagram the following actions are performed by the actors.



4.2.2 Sequence Diagram

A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

A sequence diagram shows object interactions arranged in time sequence. It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

Sequence diagrams are typically associated with use case realizations in the Logical View of the system under development. Sequence diagrams are sometimes called event diagrams or event scenarios.

If the lifeline is that of an object, it demonstrates a role. Leaving the instance name blank can represent anonymous and unnamed instances.

Messages, written with horizontal arrows with the message name written above them, display interaction. Solid arrow heads represent synchronous calls, open arrow heads represent asynchronous messages, and dashed lines represent reply messages. If a caller sends a synchronous message, it must wait until the message is done, such as invoking a subroutine. If a caller sends an asynchronous message, it can continue processing and doesn't have to wait for a response. Asynchronous calls are present in multithreaded applications, event-driven applications and in message-oriented middleware. Activation boxes, or method-call boxes, are opaque rectangles drawn on top of lifelines to represent that processes are being performed in response to the message

Objects calling methods on themselves use messages and add new activation boxes on top of any others to indicate a further level of processing. If an object is destroyed (removed from memory), an X is drawn on bottom of the lifeline, and the dashed line ceases to be drawn below it. It should be the result of a message, either from the object itself, or another.

A message sent from outside the diagram can be represented by a message originating from a filled-in circle (found message in UML) or from a border of the sequence diagram (gate in UML).

UML has introduced significant improvements to the capabilities of sequence diagrams. Most of these improvements are based on the idea of interaction fragments which represent smaller pieces of an enclosing interaction. Multiple interaction fragments are combined to create a

variety of combined fragments, which are then used to model interactions that include parallelism, conditional branches, optional interactions.

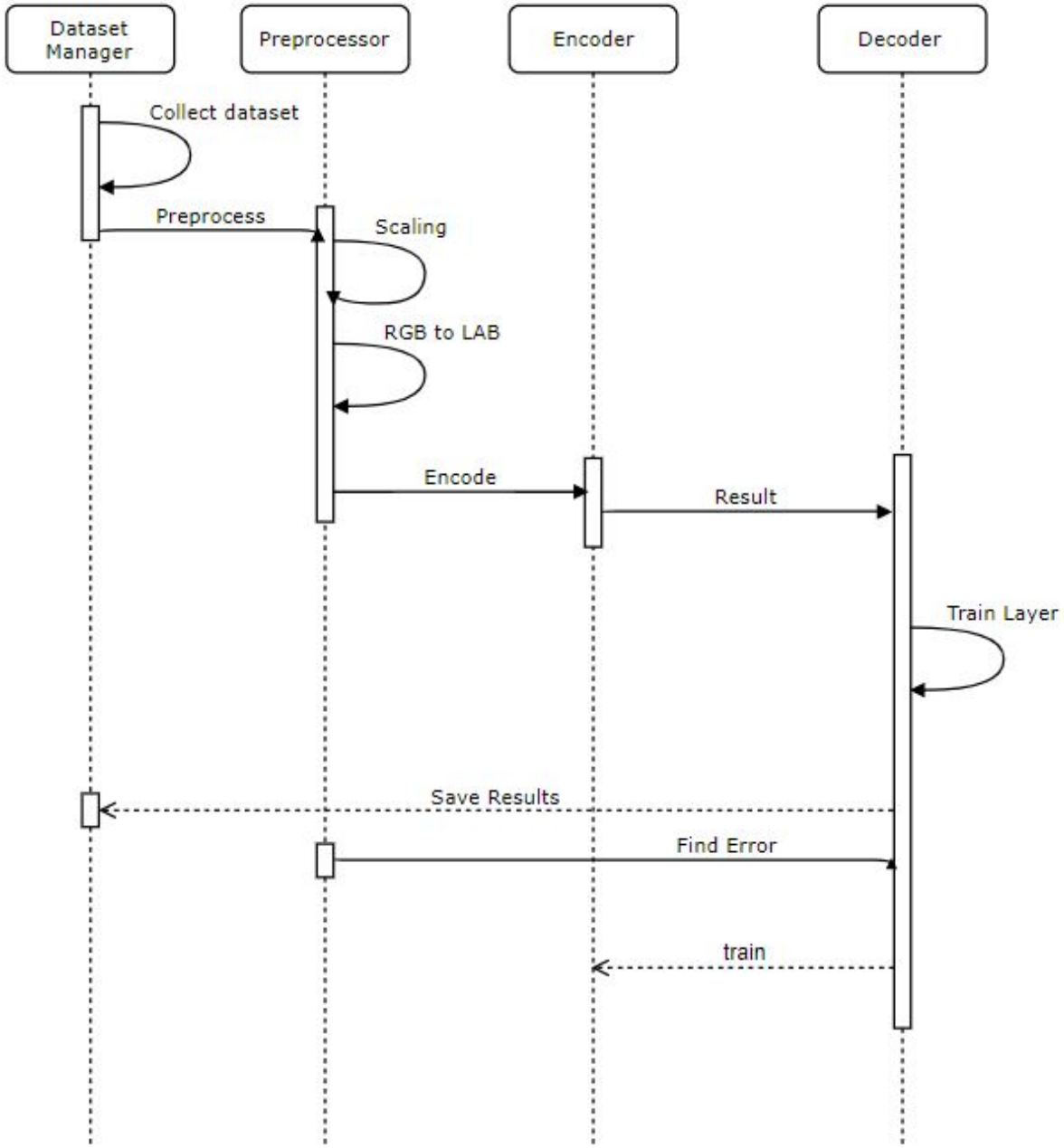
Our colorization sequence diagrams are classified into two categories:

- Training sequence diagram
- User level sequence diagram
- Application level sequence diagram

The objects used in user level sequence diagram are:

- User
- Colorizer

4.2.2.1 Training Sequence Diagram:

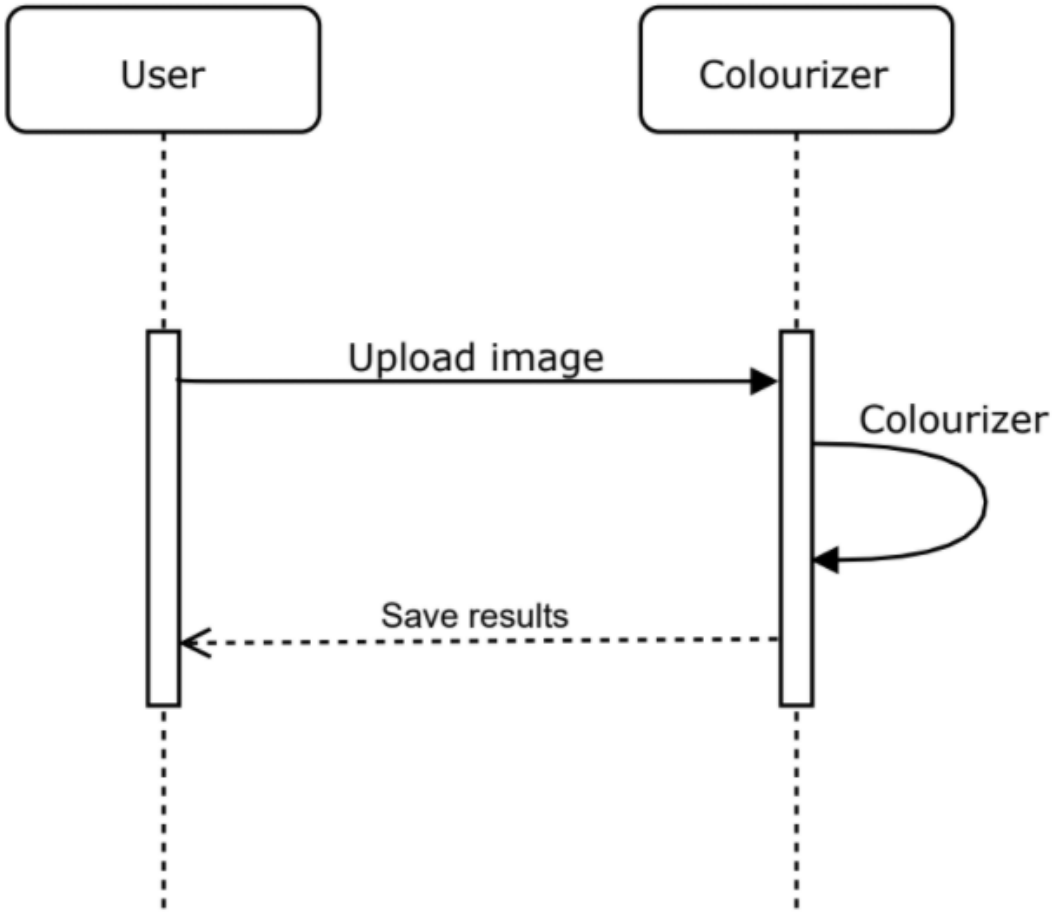


The process in the training sequence diagram is as follows:

- The user gathers the images required to train the model.
- The preprocessor will scale the image and convert RGB to Lab.
- The neural net encodes and decodes the features from the image.

- The neural net will output possible colors as a result.
- The obtained and original colors are evaluated for accuracy.

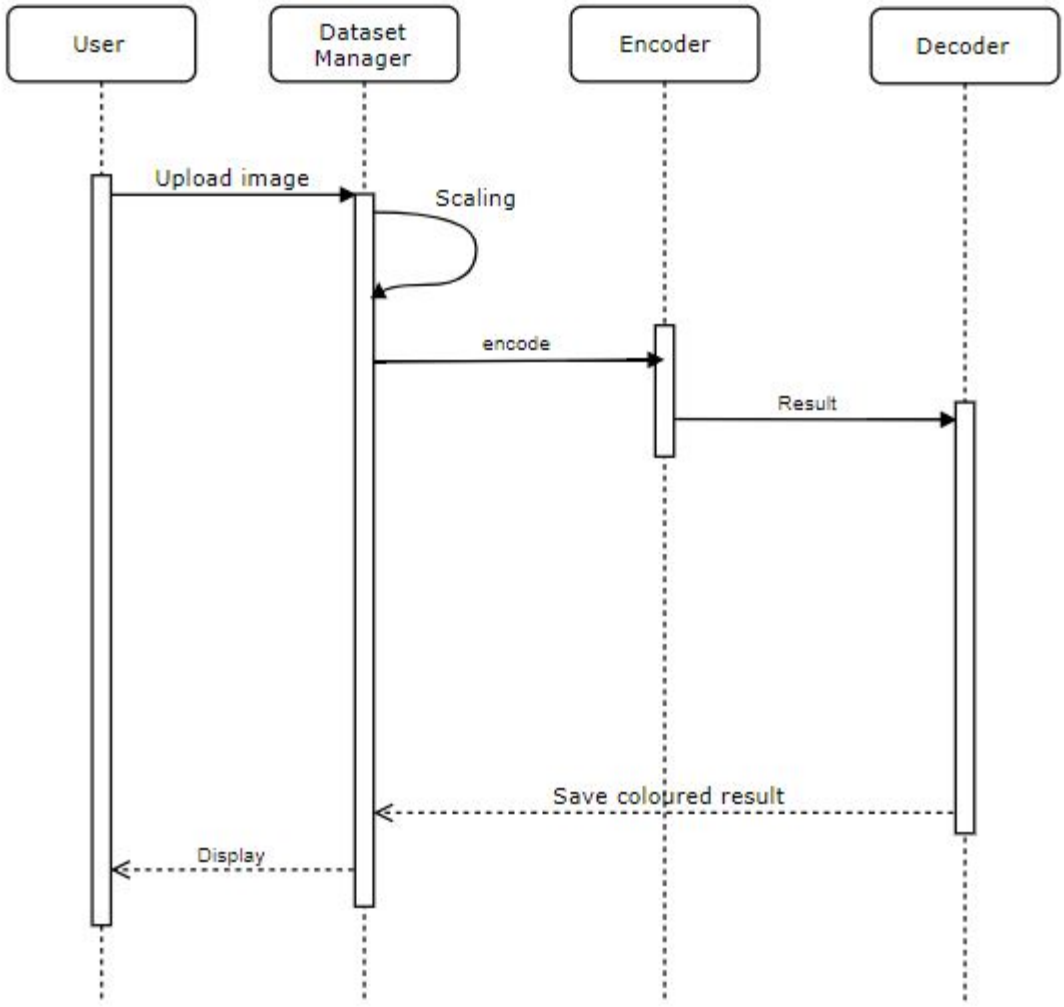
4.2.2.2 User Level Sequence Diagram:



The process in the user level sequence diagram is as follows:

- The user uploads the black and white image to the application
- Image is uploaded on to Colorization code
- Colorization detects the colors from the image
- The coloured image is then saved and displayed to the user

4.2.2.3 Application level sequence diagram:



The process in the application level sequence diagram is as follows:

- The user uploads the black and white image to the application.
- Image is uploaded on to Colorization code
- The image get scaled to a proper resolution (size)
- The neural net encodes and decodes the features from the image.
- The neural net will output possible colors as a result.
- The coloured image is then saved and displayed to the user

4.2.3 DFD Diagram

A data flow diagram (DFD) is a graphical representation of the "flow" of data through an information system, modelling its process aspects. A DFD is often used as a preliminary step to create an overview of the system without going into great detail, which can later be elaborated. DFDs can also be used for the visualization of data processing (structured design).

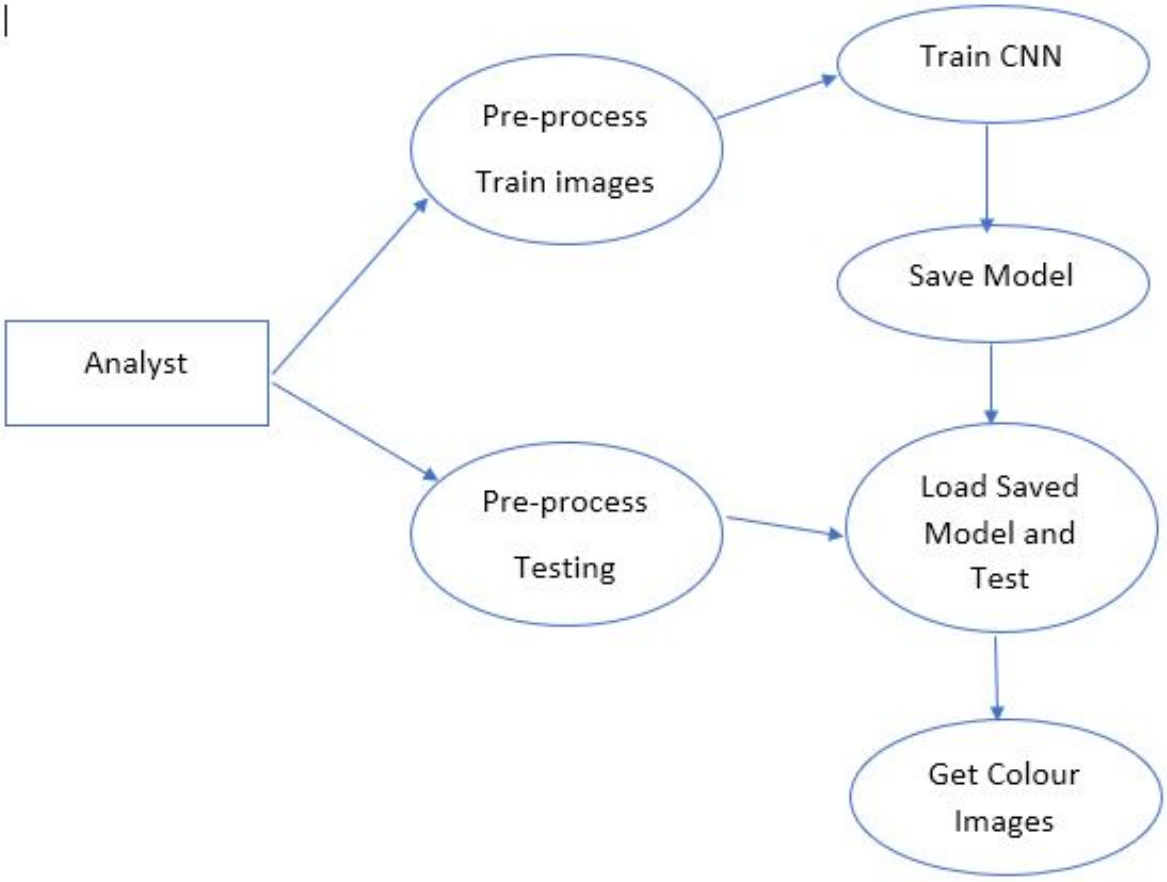
A DFD shows what kind of information will be input to and output from the system, how the data will advance through the system, and where the data will be stored. It does not show information about process timing or whether processes will operate in sequence or in parallel, unlike a traditional structured flowchart which focuses on control flow, or a UML activity workflow diagram, which presents both control and data flows as a unified model.

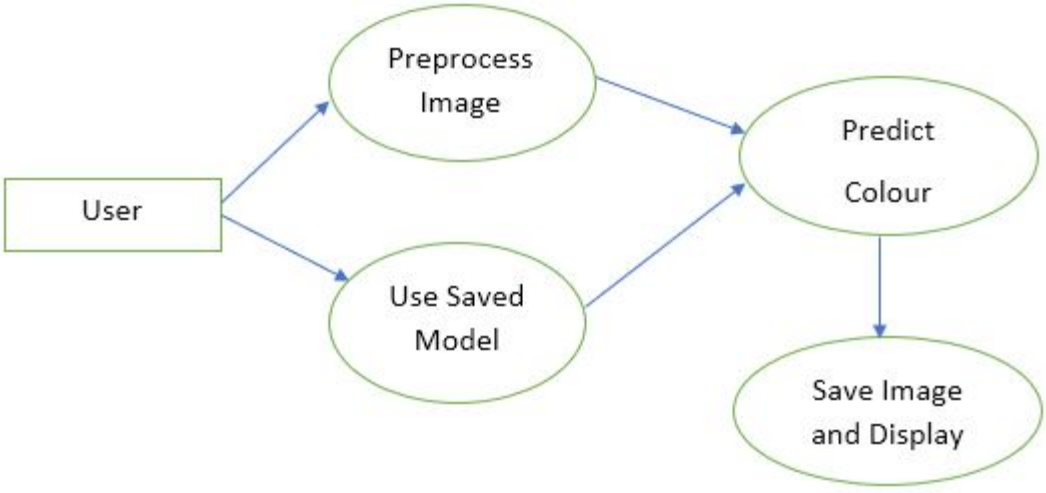
Data flow diagrams can be used in both the Analysis and Design phases of the SDLC.

There are different notations to draw data flow diagrams (Yourdon & Coad and Gane & Sarson), defining different visual representations for processes, data stores, data flow, and external entities.

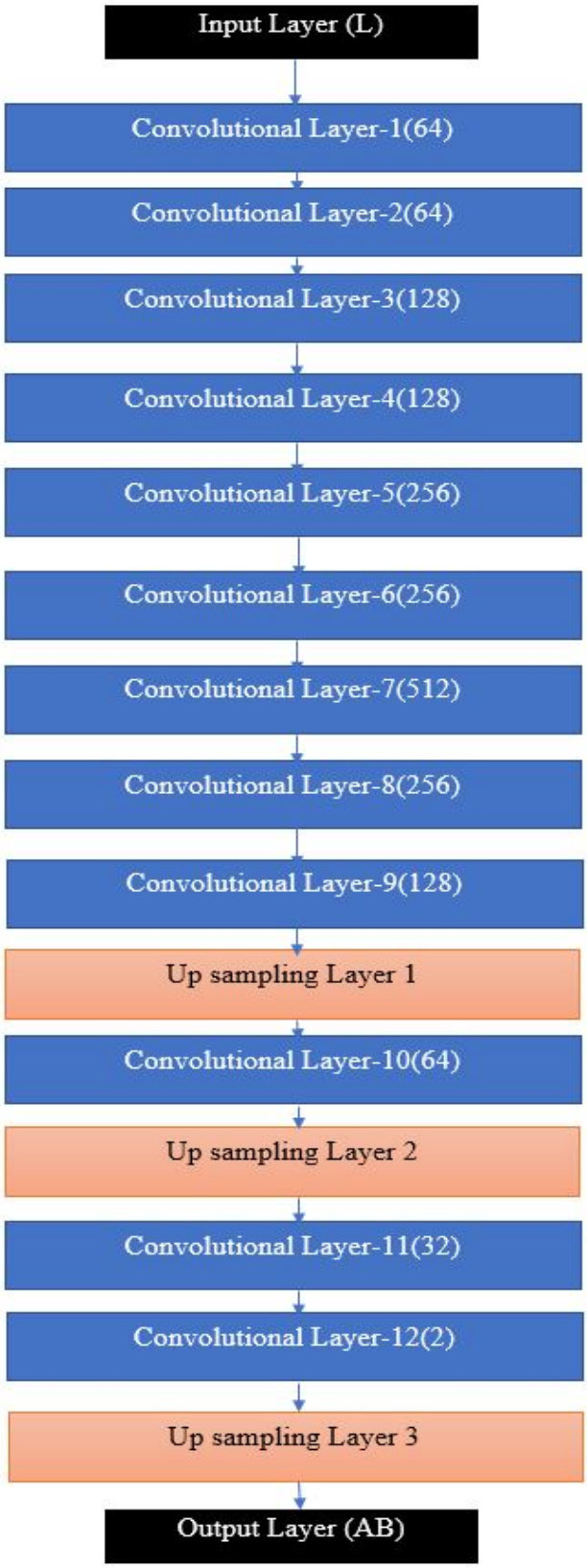
The DFD serves two purposes:

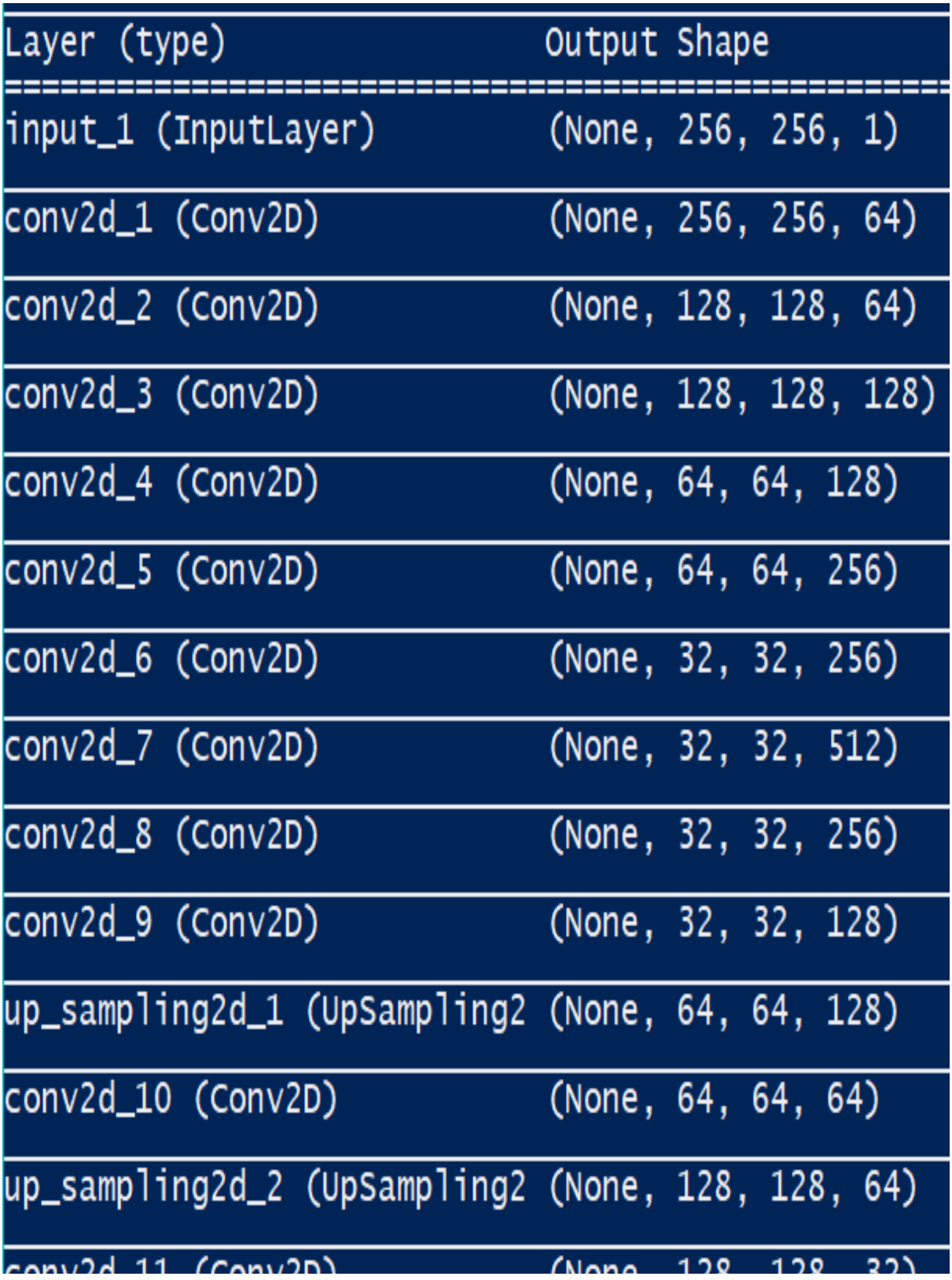
- 1. To provide an indication of how data are transformed as they move through the system.
- 2. To depict the function and sub-functions that transforms the data.


Train CNN
Pre-process
Train images
Save Model
Analyst
Pre-process
Testing
Load Saved
Model and
Test
Get Colour
Images


Preprocess
Image
Predict
Colour
User
Use Saved
Model
Save Image
and Display

4.2.4 CNN Network Architecture





Layer (type)	Output Shape
input_1 (InputLayer)	(None, 256, 256, 1)
conv2d_1 (Conv2D)	(None, 256, 256, 64)
conv2d_2 (Conv2D)	(None, 128, 128, 64)
conv2d_3 (Conv2D)	(None, 128, 128, 128)
conv2d_4 (Conv2D)	(None, 64, 64, 128)
conv2d_5 (Conv2D)	(None, 64, 64, 256)
conv2d_6 (Conv2D)	(None, 32, 32, 256)
conv2d_7 (Conv2D)	(None, 32, 32, 512)
conv2d_8 (Conv2D)	(None, 32, 32, 256)
conv2d_9 (Conv2D)	(None, 32, 32, 128)
up_sampling2d_1 (UpSampling2	(None, 64, 64, 128)
conv2d_10 (Conv2D)	(None, 64, 64, 64)
up_sampling2d_2 (UpSampling2	(None, 128, 128, 64)
conv2d_11 (Conv2D)	(None, 128, 128, 32)

Fig CNN Network Architecture when given an image of size 256*256

Convolutional Neural Networks are very similar to ordinary Neural Networks. they are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. And they still have a loss function (e.g. SVM/Softmax) on the last (fully-connected) layer.

CHAPTER-5

5. IMPLEMENTATION

5.1 Pseudo code

Pseudocode is an informal high-level description of the operating principle of a computer program or other algorithm.

It uses the structural conventions of a normal programming language, but is intended for human reading rather than machine reading. Pseudocode typically omits details that are essential for machine understanding of the algorithm, such as variable declarations, system-specific code and some subroutines. The programming language is augmented with natural language description details, where convenient, or with compact mathematical notation. The purpose of using pseudocode is that it is easier for people to understand than conventional programming language code, and that it is an efficient and environment-independent description of the key principles of an algorithm.

5.1.1 For Training:

- Specify a folder of images to train.
- Read the Images to train.
- Resize the images to 256*256.
- Convert the image to a matrix of shape (255, 255, 3).
- Convert the image from RGB to LAB colour space.
- Normalize the data.
- Split L and AB
- Pass L and AB to CNN for training.
- Save the model formed.

5.1.2 For Testing:

- Specify a folder of images to test.

- Read the Images to test.
- Resize the images to 256*256.
- Convert the image to a matrix.
- If the image is colour convert it into grayscale, if not ignore this step.
- Convert the image from RGB to LAB colour space, if the image is Colour.
- Normalize the data.
- Split L and AB, if the image is colour, otherwise directly use the L from Black and White.
- Load the Trained model.
- Pass L of the image to predict AB.
- Create an empty canvas of shape (255, 255, 3).
- Load L into the canvas.
- Load Predicted AB from the model into the canvas.
- Convert the image to RGB.
- Save the image.

5.1.3 For Prediction from Web Application:

- Open a Browser
- Go to 127.0.0.1:8000 or the ip address or the page name in which the server is running.
- Click the upload image button.
- Select an image from the file directory dialog.
- Click on Start Colouring button.

- The given Black and White image and the respective Colour image is displayed below.

5.2 Code snippets:

5.2.1 Training Code:

```
# -*- coding: utf-8 -*-


import numpy as np

import utils

from skimage.io import imsave

from skimage.color import rgb2lab, lab2rgb

from Net import Net

from keras.models import load_model

from keras.models import model_from_json


# Define Train Images Path

TRAIN_FOLDER = 'Train/'


# Get Train Data.

TRAIN_DATA = utils.get_train_data(TRAIN_FOLDER)

TRAIN_DATA_SIZE = len(TRAIN_DATA)


# Get the CNN model
```

```
net = Net(train=True)

CNN = net.encode()


# Define BatchSize

BATCH_SIZE = 50


if BATCH_SIZE < TRAIN_DATA_SIZE:

    steps = TRAIN_DATA_SIZE / BATCH_SIZE

else:

    steps = 1


###################################

# Comment next three lines while testing.

###################################

# Train model

train_batch = utils.image_l_a_b_gen(TRAIN_DATA, BATCH_SIZE)

CNN.fit_generator(train_batch, epochs=1000, steps_per_epoch=steps)


# Save model

model.save("model.h5")


###################################
```

```
# Uncomment this to test the model.

###################################

'''

# Load model

model.load("model.h5")


# Test model

TEST_FOLDER = 'Test/'

Xtest_l, Ytest_a_b = utils.get_test_data(TEST_FOLDER)

# Evaluate model

print(CNN.evaluate(Xtest_l, Ytest_a_b, batch_size=BATCH_SIZE))




# Getting a and b

test_a_b = CNN.predict(Xtest_l)

test_a_b *= 128


# Output colorizations

for i in range(len(test_a_b)):

    canvas = np.zeros((256, 256, 3))

    canvas[:, :, 0] = Xtest_l[i][:, :, 0]

    canvas[:, :, 1:] = test_a_b[i]
```

```
    imsave("img_"+str(i)+".png", lab2rgb(cur))
'''
```

5.2.2 Testing Code:

```
import tensorflow as tf
import cv2
from app.tf.utils import *
from app.tf.net import Net
from skimage.io import imsave
from skimage.transform import resize

def colorize():
    """
    This method reads in a black and white image and converts
    it to colour image.
    """
    # Read the B/W image.
    img = cv2.imread('static/gray.jpg')
    if len(img.shape) == 3:
        img = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    img = img[None, :, :, None]
```

```
    data_l = (img.astype(dtype=np.float32)) / 255.0 * 100 - 50


    # As we are testing train should be false.
    autocolor = Net(train=False)
    # encode returns a loaded_model
    model = autocolor.encode(data_l)
    # decode returns a RGB image
    img_rgb = decode(data_l, model)


   # Save the Colour image.
   imsave('static/color.jpg', img_rgb)
```

5.2.3 Data Structures:

```
# -*- coding: utf-8 -*-
from keras.layers import Conv2D, UpSampling2D, InputLayer
from keras.models import Sequential
from keras.models import load_model


class Net:
    def __init__(self, train=True):
        self.train=train


    def encode(self):
```

```
    if self.train == True:
        return self.net()
    else:
        return self.loaded_model()

def net(self):
    """
    This method returns a CNN for training.
    """
    model = Sequential()
    model.add(InputLayer(input_shape=(None, None, 1)))
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same', strides=2))
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same', strides=2))
    model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(256, (3, 3), activation='relu', padding='same', strides=2))
    model.add(Conv2D(512, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
    model.add(UpSampling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
```

```
        model.add(UpSampling2D((2, 2)))
        model.add(Conv2D(32, (3, 3), activation='relu', padding='same'))
        model.add(Conv2D(2, (3, 3), activation='tanh', padding='same'))
        model.add(UpSampling2D((2, 2)))
        model.compile(optimizer='rmsprop', loss='mse')
        return model
    def loaded_model(self):
        """
        This method returns a loaded_model.
        """
        return load_model("model.h5")
```

CHAPTER-6

6. TESTING

Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not.

Testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

For testing the accuracy of the system, we used, in a first test scenario, an image which contained 256x256 resolution only . The construction of the training set, which consisted of two images containing 40 examples of each 256x256. We reproduced it and documented the process. First off, let's look at some of the results/failures from our experiments

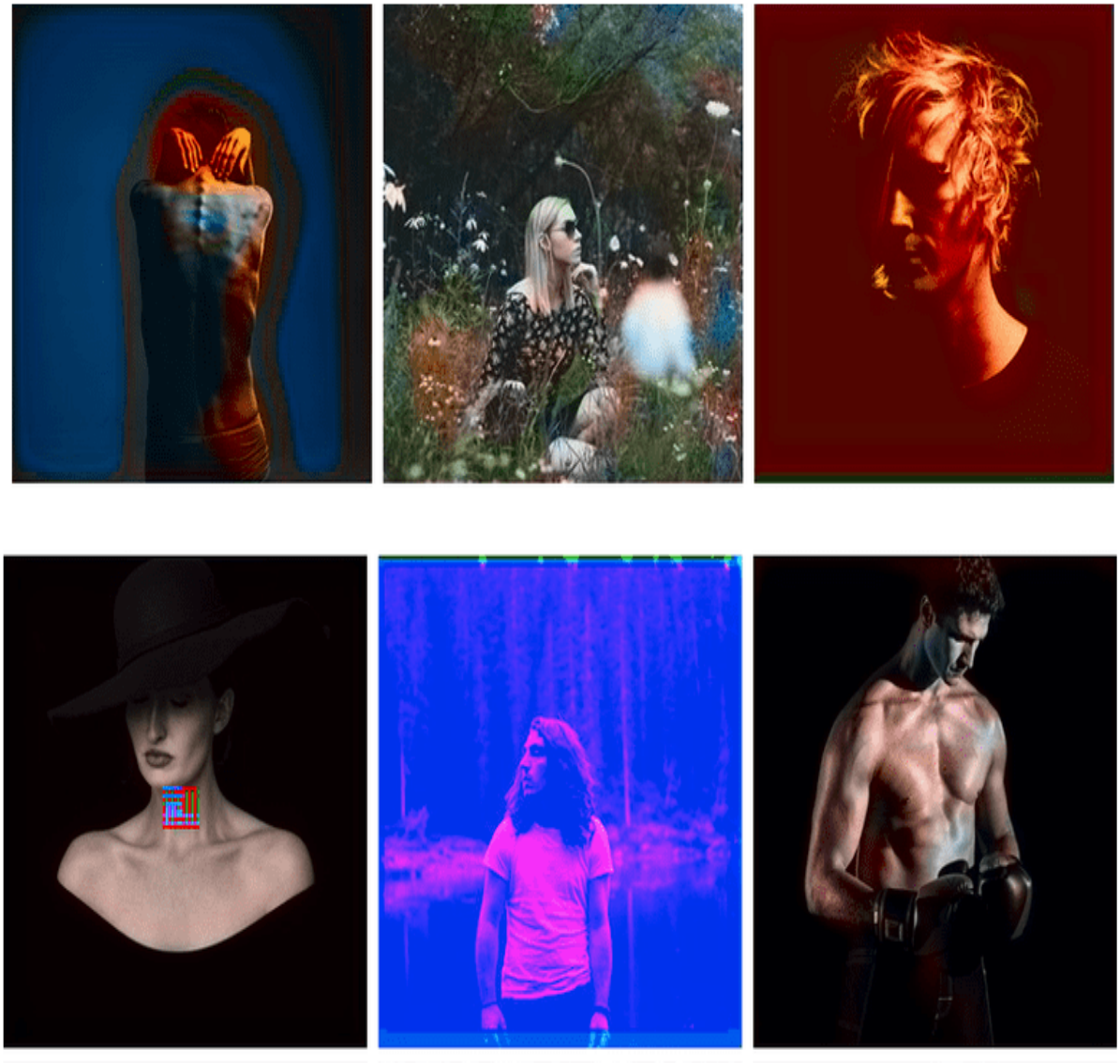


Fig. 6.1 Experimental Results 1

For the next test scenario we used for training only the features corresponding to a single image. The image used for testing is the same image, and the construction of the training set, which consisted of one image containing 100 epochs took 19.5075 sec. Given a grayscale Macbeth color chart as input, the network was unable to recover its colors. The network is trained by freezing the representation up to certain points, and fine-tuning the remainder. The results are presented below.

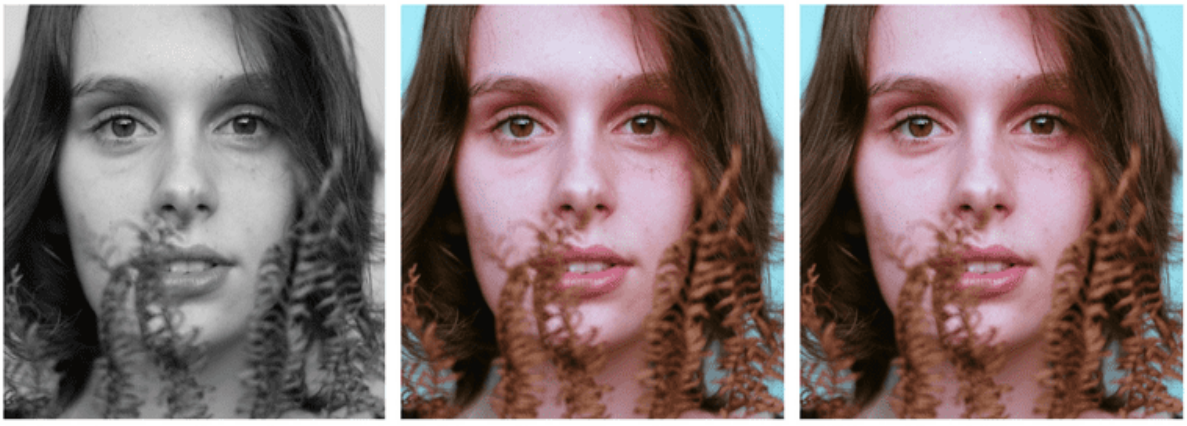


Fig. 6.2 Experimental Results 2

evaluating the perceptual realism of our colorizations, along with other measures of accuracy. We compare our full algorithm to several variants, along with recent and concurrent work. We test colorization as a method for self supervised representation learning. Without accounting for this, the loss function is dominated by desaturated ab values. We account for

the class imbalance problem by reweighting the loss of each pixel at train time based on the pixel color rarity. Finally, we show qualitative examples on legacy black and white images.

White these were the experimental results from our initial model testing phase, we have then trained the model with a higher number of input images and epochs. The results that were observed we better.

In the first case, we trained 10 landscape images for 1000 epochs. Then we tried to test it with a landscape image it has never seen before. We did the same with 10 female faces for 2000 epochs. The result is shown below.

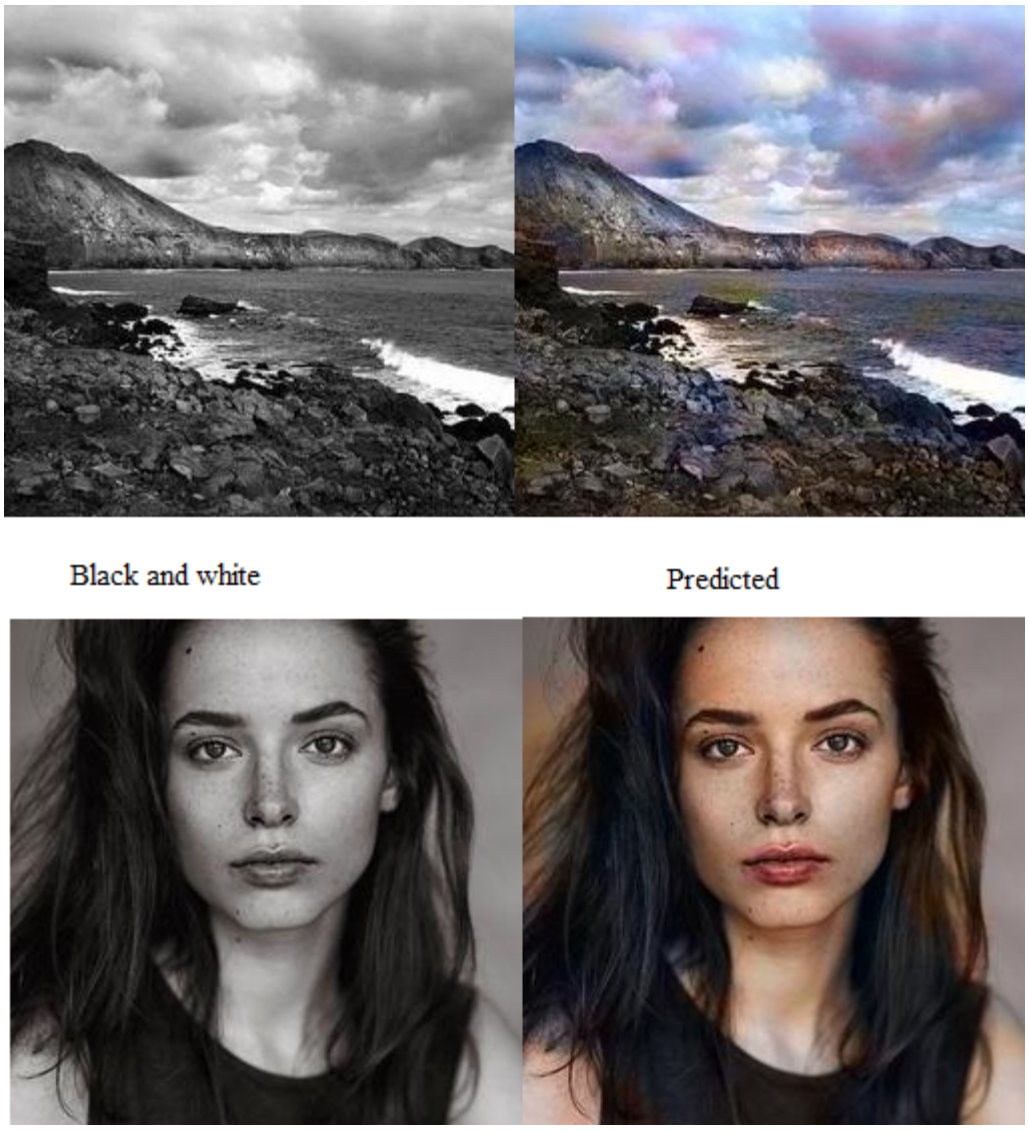
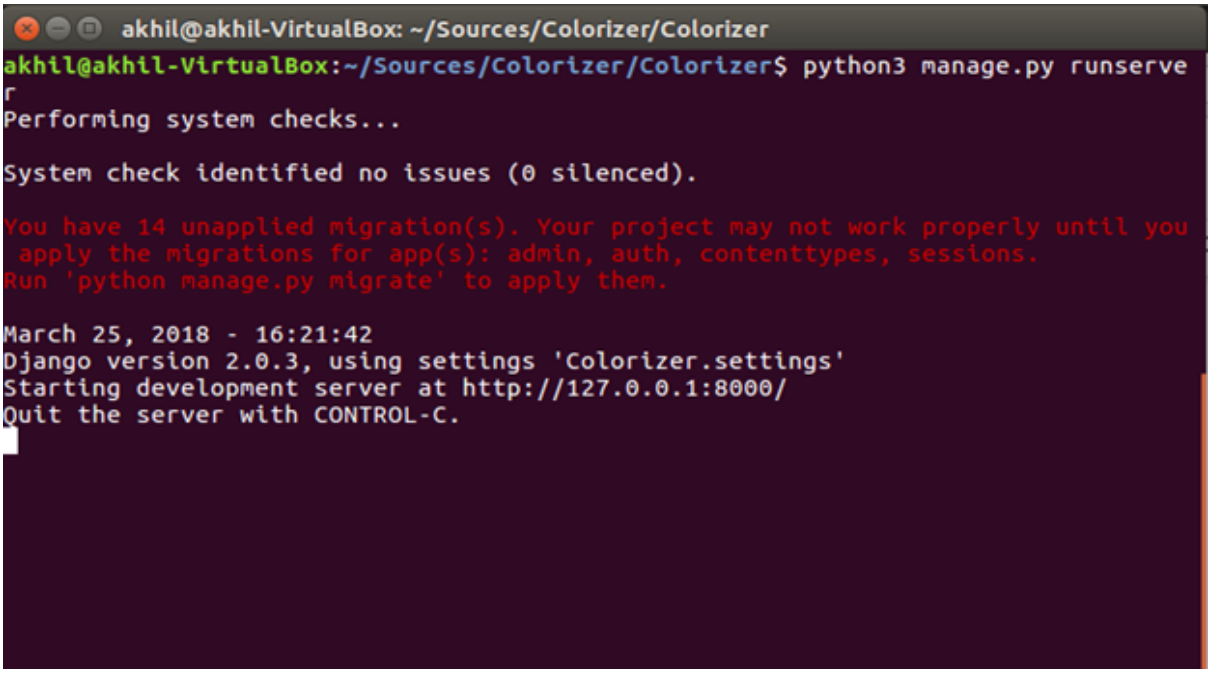


Fig 6.3 Experimental Results 3

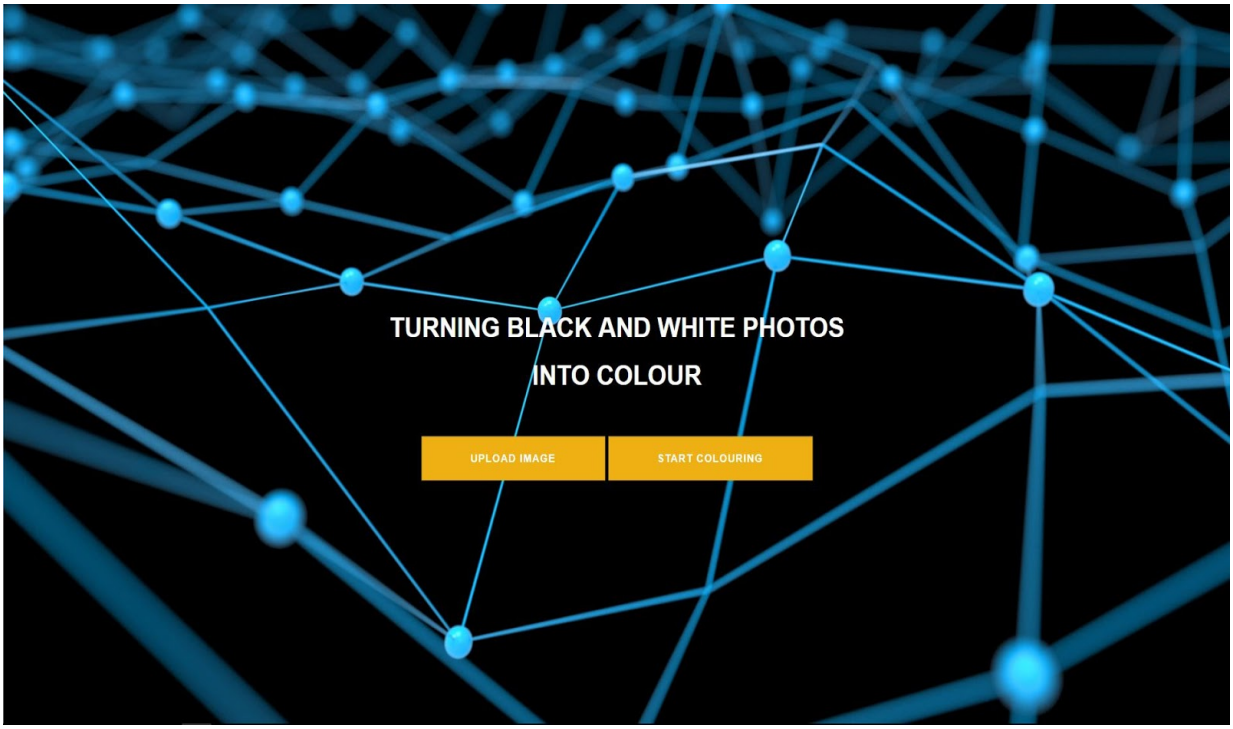
CHAPTER-7

7. SCREENSHOTS

7.1 Running the web server



7.2 Main page



BLACK AND WHITE IMAGE

COLOURED IMAGE

PREVIOUS PREDICTIONS


Black And White


Coloured


Black And White


Coloured

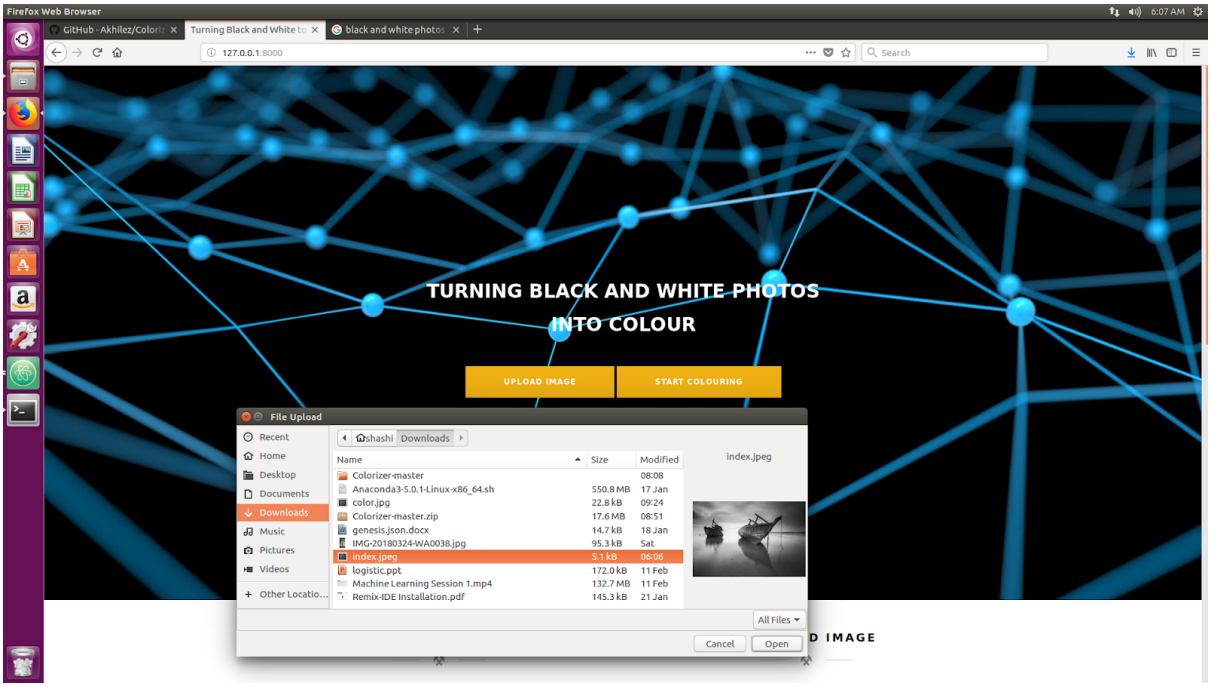

Black And White5


Coloured

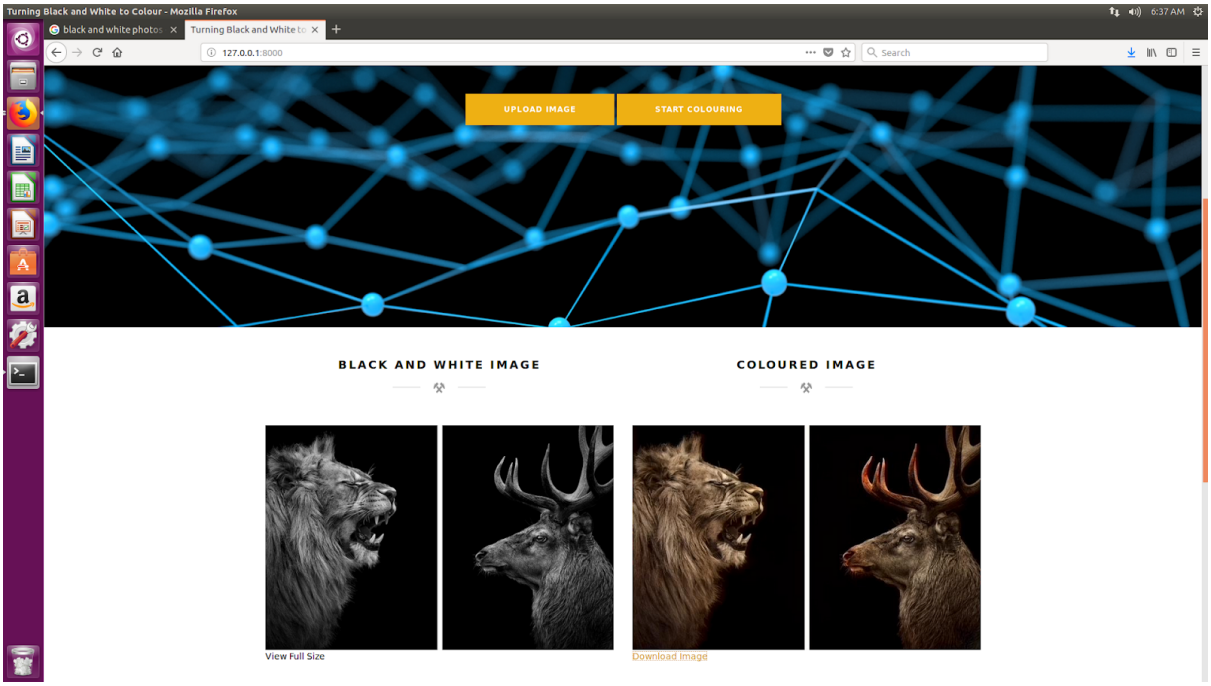

Black And White


Coloured

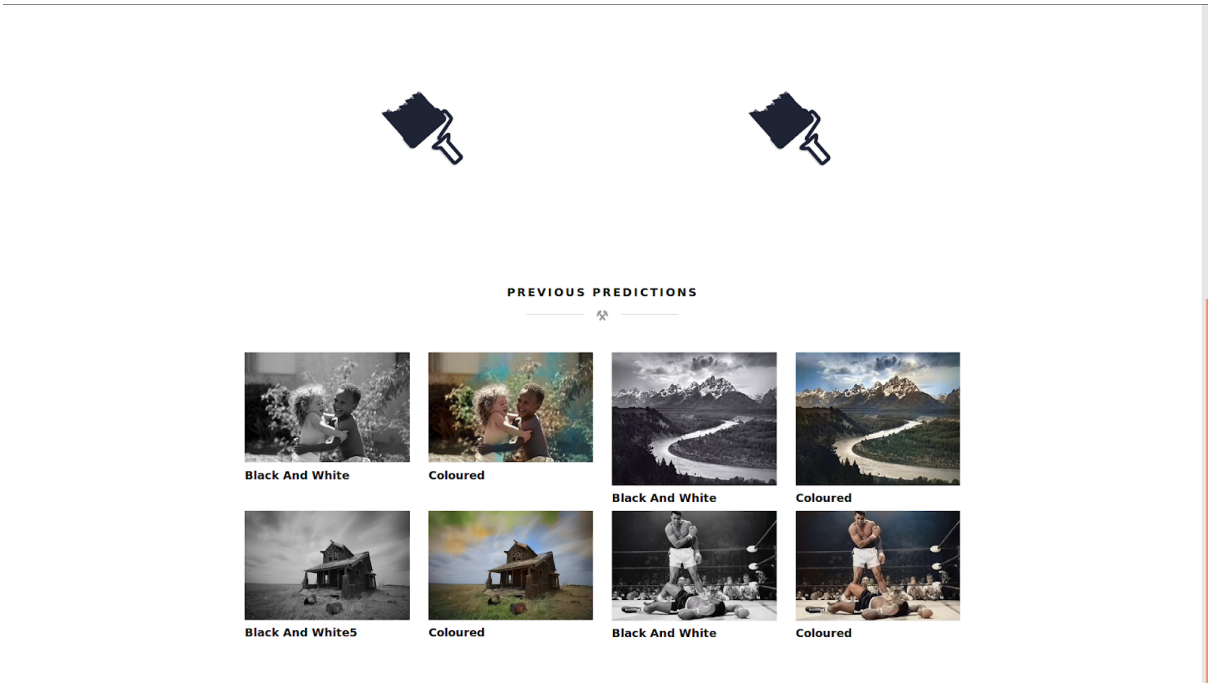
7.3 Upload image

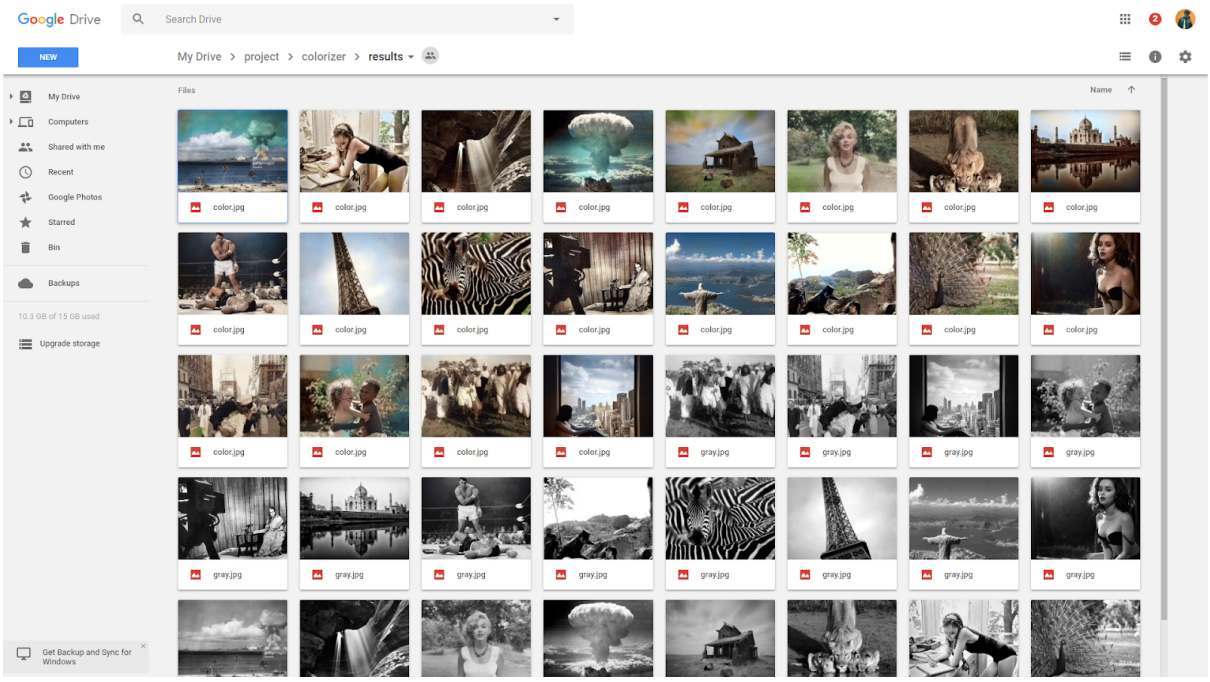


7.4 Obtained Results

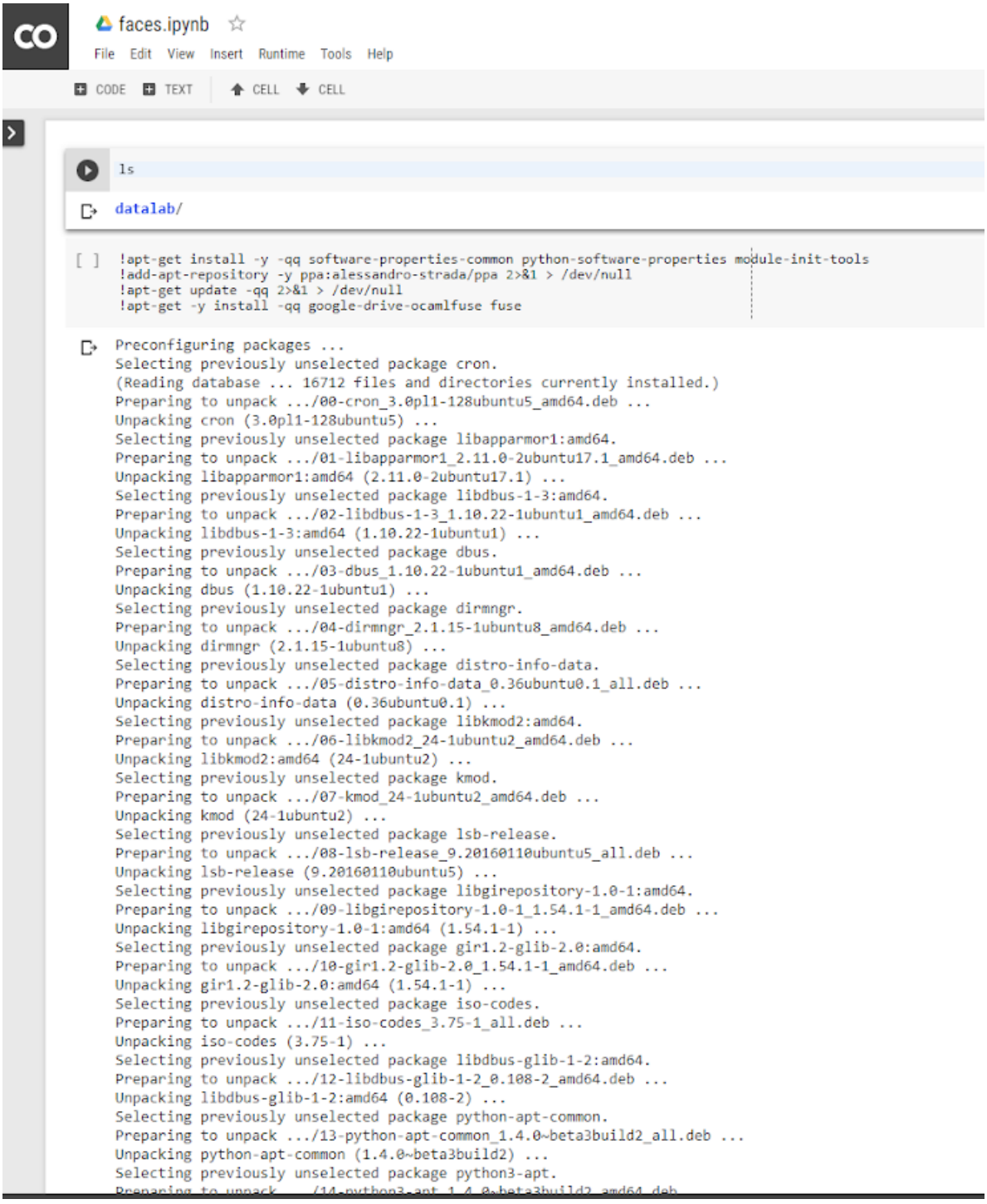


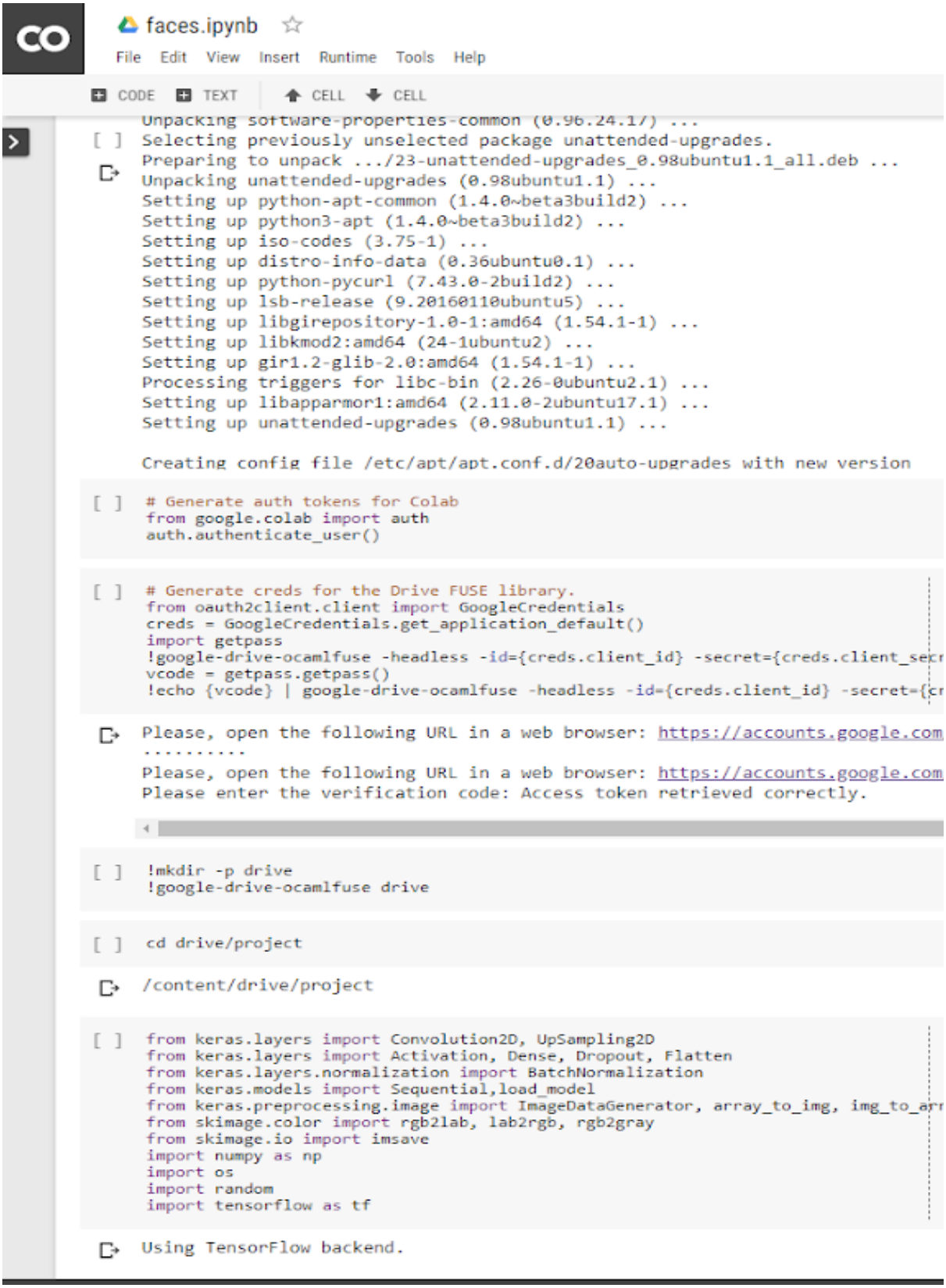
7.5 Previous Results





7.6 Training Screenshots




faces.ipynb
File Edit View Insert Runtime Tools Help
CODE TEXT CELL CELL
Unpacking software-properties-common (0.96.24.17) ...
Selecting previously unselected package unattended-upgrades.
Preparing to unpack .../23-unattended-upgrades_0.98ubuntu1.1_all.deb ...
Unpacking unattended-upgrades (0.98ubuntu1.1) ...
Setting up python-apt-common (1.4.0~beta3build2) ...
Setting up python3-apt (1.4.0~beta3build2) ...
Setting up iso-codes (3.75-1) ...
Setting up distro-info-data (0.36ubuntu0.1) ...
Setting up python-pycurl (7.43.0-2build2) ...
Setting up lsb-release (9.20160110ubuntu5) ...
Setting up libgirepository-1.0-1:amd64 (1.54.1-1) ...
Setting up libkmod2:amd64 (24-1ubuntu2) ...
Setting up gir1.2-glib-2.0:amd64 (1.54.1-1) ...
Processing triggers for libc-bin (2.26-0ubuntu2.1) ...
Setting up libapparmor1:amd64 (2.11.0-2ubuntu17.1) ...
Setting up unattended-upgrades (0.98ubuntu1.1) ...
Creating config file /etc/apt/apt.conf.d/20auto-upgrades with new version
# Generate auth tokens for Colab
from google.colab import auth
auth.authenticate_user()
# Generate creds for the Drive FUSE library.
from oauth2client.client import GoogleCredentials
creds = GoogleCredentials.get_application_default()
import getpass
!google-drive-ocamlfuse -headless -id={creds.client_id} -secret={creds.client_sec
vcode = getpass.getpass()
!echo {vcode} | google-drive-ocamlfuse -headless -id={creds.client_id} -secret={cr
Please, open the following URL in a web browser: https://accounts.google.com
.....
Please, open the following URL in a web browser: https://accounts.google.com
Please enter the verification code: Access token retrieved correctly.
!mkdir -p drive
!google-drive-ocamlfuse drive
cd drive/project
/content/drive/project
from keras.layers import Convolution2D, UpSampling2D
from keras.layers import Activation, Dense, Dropout, Flatten
from keras.layers.normalization import BatchNormalization
from keras.models import Sequential,load_model
from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_arr
from skimage.color import rgb2lab, lab2rgb, rgb2gray
from skimage.io import imsave
import numpy as np
import os
import random
import tensorflow as tf
Using TensorFlow backend.

faces.ipynb ☆

File Edit View Insert Runtime Tools Help

CODE TEXT CELL CELL

```
[ ]  !mkdir -p drive
     !google-drive-ocamlfuse drive
```

```
[ ]  cd drive/project
```

```
/content/drive/project
```

```
[ ]  from keras.layers import Convolution2D, UpSampling2D
     from keras.layers import Activation, Dense, Dropout, Flatten
     from keras.layers.normalization import BatchNormalization
     from keras.models import Sequential,load_model
     from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_a
     from skimage.color import rgb2lab, lab2rgb, rgb2gray
     from skimage.io import imsave
     import numpy as np
     import os
     import random
     import tensorflow as tf
```

```
Using TensorFlow backend.
```

```
[ ]  # Image transformer
     datagen = ImageDataGenerator(
         rescale=1.0/255,
         shear_range=0.2,
         zoom_range=0.2,
         horizontal_flip=True)
```

```
[ ]  # Get images
     X = []
     for filename in os.listdir('Train'):
       X.append(img_to_array(load_img('Train/'+filename)))
     X = np.array(X)

     # Set up train and test data
     split = int(0.9*len(X))
     Xtrain = X[:split]
     Xtest = rgb2lab(1.0/255*X[split:])[:,:,:,0]
     Xtest = Xtest.reshape(Xtest.shape+(1,))
     Ytest = rgb2lab(1.0/255*X[split:])[:,:,:,1:]

     # Set up model
     N = 5
     model = Sequential()
     num_maps1 = [4, 8, 16, 32, 64]
     num_maps2 = [8, 16, 32, 64, 128]

     # Convolutional layers
     for i in range(N):
       if i == 0:
         model.add(Convolution2D(num_maps1[i], 3, 3, border_mode='same', subsample=(2
       else:
         model.add(Convolution2D(num_maps1[i], 3, 3, border_mode='same', subsample=(2
       model.add(BatchNormalization())
       model.add(Activation('relu'))
       model.add(Convolution2D(num_maps2[i], 3, 3, border_mode='same', subsample=(1,
       model.add(BatchNormalization())
       model.add(Activation('relu'))

     # Upsampling layers
     for i in range(N):
       model.add(UpSampling2D(size=(2, 2)))
```

```
model.save("model.h5")

# Test model
print model.evaluate(Xtest, Ytest, batch_size=batch_size)
output = model.predict(Xtest)

# Output colorizations
for i in range(len(output)):
  cur = np.zeros((128, 128, 3))
  cur[:,:,0] = Xtest[i][:,:,0]
  cur[:,:,1:] = output[i]
  imsave("img_"+str(i)+".png", lab2rgb(cur))
  imsave("img_gray_"+str(i)+".png", rgb2gray(lab2rgb(cur)))
```

```
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:22: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(4, (3, 3), pac
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:27: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(8, (3, 3), pac
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:24: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(8, (3, 3), pac
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:27: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(16, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:24: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(16, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:27: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(32, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:24: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(32, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:27: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(64, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:24: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(64, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:27: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(128, (3, 3), p
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:34: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(128, (3, 3), p
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:38: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(64, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:34: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(64, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:38: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(32, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:34: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(32, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:38: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(16, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:34: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(16, (3, 3), pa
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:38: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(8, (3, 3), pac
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:34: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(8, (3, 3), pac
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:42: UserWarning: Update your `Conv2D` call to the Keras 2 API: `Conv2D(2, (3, 3), pac
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:61: UserWarning: The semantics of the Keras 2 argument `steps_per_epoch` is not the s
/usr/local/lib/python2.7/dist-packages/ipykernel_launcher.py:61: UserWarning: Update your `fit_generator` call to the Keras 2 API: `fit_generator(
Epoch 1/15
500/500 [==============================] - 1787s 4s/step - loss: 42.8147
Epoch 2/15
366/500 [====================>.........] - ETA: 7:55 - loss: 12.1781500/500 [==============================] - 1775s 4s/step - loss: 11.6616
Epoch 3/15
500/500 [==============================] - 1768s 4s/step - loss: 8.8609
Epoch 4/15
 62/500 [==>...........................] - ETA: 25:39 - loss: 7.8092500/500 [==============================] - 1759s 4s/step - loss: 7.4569
Epoch 5/15
466/500 [==========================>...] - ETA: 1:59 - loss: 6.7163500/500 [==============================] - 1759s 4s/step - loss: 6.6962
Epoch 6/15
500/500 [==============================] - 1775s 4s/step - loss: 6.1229
Epoch 7/15
 94/500 [====>.........................] - ETA: 24:03 - loss: 5.9050500/500 [==============================] - 1775s 4s/step - loss: 5.6980
Epoch 8/15
480/500 [===========================>..] - ETA: 1:11 - loss: 5.3805500/500 [==============================] - 1781s 4s/step - loss: 5.3702
Epoch 9/15
```

```
500/500 [==============================] - 1768s 4s/step - loss: 8.8609
Epoch 4/15
 62/500 [==>...........................] - ETA: 25:39 - loss: 7.8092500/500 [======
Epoch 5/15
466/500 [==========================>...] - ETA: 1:59 - loss: 6.7163500/500 [=======
Epoch 6/15
500/500 [==============================] - 1775s 4s/step - loss: 6.1229
Epoch 7/15
 94/500 [====>.........................] - ETA: 24:03 - loss: 5.9050500/500 [======
Epoch 8/15
480/500 [===========================>..] - ETA: 1:11 - loss: 5.3805500/500 [=======
Epoch 9/15
500/500 [==============================] - 1788s 4s/step - loss: 5.0592
Epoch 10/15
 98/500 [====>.........................] - ETA: 24:08 - loss: 5.0398500/500 [======
Epoch 11/15
479/500 [===========================>..] - ETA: 1:14 - loss: 4.7669500/500 [=======
Epoch 12/15
500/500 [==============================] - 1769s 4s/step - loss: 4.6174
Epoch 13/15
 95/500 [====>.........................] - ETA: 23:46 - loss: 4.6121500/500 [======
Epoch 14/15
477/500 [===========================>..] - ETA: 1:21 - loss: 4.3865500/500 [=======
Epoch 15/15
500/500 [==============================] - 1769s 4s/step - loss: 4.2660
47/47 [==============================] - 1s 28ms/step
12.610876083374023
/usr/local/lib/python2.7/dist-packages/skimage/util/dtype.py:122: UserWarning: Poss
  .format(dtypeobj_in, dtypeobj_out))
/usr/local/lib/python2.7/dist-packages/skimage/util/dtype.py:122: UserWarning: Poss
  .format(dtypeobj_in, dtypeobj_out))
```

8. Conclusion

Colorizing images is a deeply fascinating problem. It is as much as a scientific problem as artistic one. Through this project, we have proven that turning black and white images to color images is possible through CNN. With enough color images as training data, we are able to train the model that can output color image with decent accuracy. We have observed that when the training model is designed well and training data is diverse enough, then the classification becomes less of a factor. This means that the produced color image is more accurate than those produced by the other models and is also faster.

9. Future Enhancements

- Fuse the convolutional layers with a classifier to eliminate the classification problem. This may improve the output accuracy.
- Implementing it with a pre-trained model may improve the results.
- Using a different dataset than the ones trained for our project may improve the results.
- Enable the network to grow in accuracy with more pictures
- Build an amplifier within the RGB color space. Create a similar model to the coloring network, that takes a saturated colored image as input and the correct colored image as output.
- Implement a weighted classification
- Use a classification neural network as a loss function. Pictures that are classified as fake produce an error. It then decides how much each pixel contributed to the error.
- Apply it to video. Don’t worry too much about the colorization, but make the switch between images consistent. You could also do something similar for larger images, by tiling smaller ones.

10. References

- *Colorizing B&W Photos with Neural Networks - Emil Wallner on October 13, 2017*
- *Colorful Image Colorization Richard Zhang, Phillip Isola, Alexei A. Efros ECCV 5th October 2015*
- *Dahl, R.: Automatic colorization. In: http://tinyclouds.org/colorize/. (2016)*
- *Charpiat, G., Hofmann, M., Scholkopf, B.: Automatic image colorization via multimodal predictions. In: Computer Vision–ECCV 2008. Springer (2008) 126–139*
- *Ramanarayanan, G., Ferwerda, J., Walter, B., Bala, K.: Visual equivalence: towards a new standard for image fidelity. ACM Transactions on Graphics (TOG) 26(3) (2007) 76*
- *Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014)*

11. List of Abbreviations

CNN:- Convolutional Neural Networks

SSD:- Solid State Drive

GPU:- Graphics Processing Unit

RGB:- Red Green Blue

UML:- Unified Modelling Language

DFD:- Data Flow Diagram