NITTE EDUCATION TRUST | NMAM INSTITUTE OF TECHNOLOGY

AI COLOURISER

Colorization of black-and-white images using Opencv and Deep Learning

DECEEMBER 2022

SUBMITTED TO: SHWETHA G K

RYAN D'SOUZA(4NM20CS148)

GAURAV AJILA(4NM20CS112)

CONTENTS:

ABSTRACT:

Given a grayscale photograph as input, this paper attacks the problem of hallucinating a *plausible* colour version of the photograph. This problem is clearly under constrained, so previous approaches have either relied on significant user interaction or resulted in desaturated colorizations. We propose a fully automatic approach that produces vibrant and realistic colorizations. We embrace 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 colour images. We evaluate our algorithm using a "colorization Turing test," asking human participants to choose between a generated and ground truth colour image. Our method successfully fools humans on 32% of the trials, significantly higher than previous methods. Moreover, we show 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.

INTRODUCTION:

In this thesis, we concern ourselves with the possibility of automated grayscale image colorization using deep convultional neural networks, trained on and applied to a unique cartoon dataset. Generally, the idea of coloring a grayscale image is a task that is simple for the human mind, we learn from an early age to fill in missing colours in coloring books, by remembering that grass is green, the sky is blue with white clouds or that an apple can be red or green. In some domains, automatic colorization can be very useful even without semantic understanding of the image, the simple act of adding colour can increase the amount of information that we gather from an image. For example, it is commonly used in medical imaging to improve visual quality when viewed by human eye. Majority of equipment used for medical imaging captures grayscale images, and these images may contain parts that are difficult to interpret, due to inability of the eye of an average person to distinguish more than a few hues of gray.

As a computer vision task, several user-assisted methods have been proposed for colourization, be it for natural or hand-drawn images, and expect supplied localized colour hints, or provided reference images that are semantically similar to the target image that we can transfer colour from, or even just keywords describing the image, to search the web for the reference images automatically. However, the high-level understanding of scene composition and object relations required for

colorization of more complex images remains the reason developing new, fully automated solutions is problematic. Recently, the approach to this task has shifted significantly from the human-assisted methods to fully automated solutions, implemented predominantly by convolutional neural networks.

Research in this area seeks to make automated colorization cheaper and less time consuming, and by that, allowing its application on larger scale. With the advent of deep convolutional neural networks, the task has been getting increased amounts of attention as a representative issue for complete visual understanding of artificial intelligence, similar to what many thought object recognition to be previously, since to truly convincingly colorize a target image, the method needs to be able to correctly solve a number of subtasks similar to segmentation, classification and localization.

LITERATURE REVIEW:

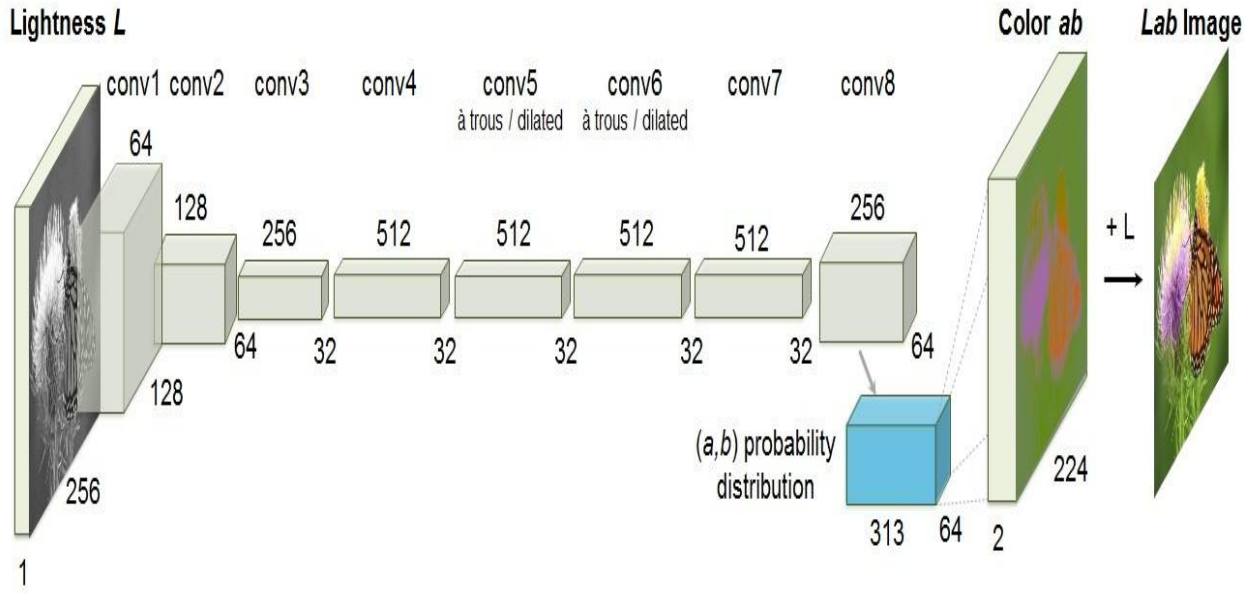
Our goal is not necessarily to recover the actual ground truth colour, but rather to produce a plausible colorization that could potentially fool a human observer. Therefore, our task becomes much more achievable: to model enough of the statistical dependencies between the semantics and the textures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

Colorization of grayscale images has become a more researched area in the recent years, thanks to the advent of deep convolutional neural networks. We attempt to apply this concept to colorization of cartoon images obtained from video sequences. Previous similar research focused mainly colorization of natural images, while colorization of cartoons is traditionally done by leveraging manual scribble methods. Our proposed method is a fully automated process. To implement it, we propose and compare two distinct convolutional neural network architectures trained under various loss functions. We aim to compare each variant based on results obtained as individual images and videos.

RECENT RELATED WORK

We train a CNN to map from a grayscale input to a distribution over quantized colour value outputs using the architecture given below. In the following, they focus on

the design of the objective function, and their technique for inferring point estimates of colour from the predicted colour distribution.



The scheme is based on the colorization technique which can colorize a monochrome image by giving a small number of colour pixels. We develop algorithms useful for colour image coding. First, the luminance component is separated from an input colour image. Then, a small number of colour seeds are selected as chrominance information. The luminance image component is coded by a lossy coding technique and the chrominance image component is stored as colour seeds. The decoding is performed by the colorization algorithm. It is shown that this colorization technique is effective to image coding, especially for high compression rate, through the experiments using different types of images. We develop a new strategy that attempts to account for the higher-level context of each pixel. The colorizations generated by our

approach exhibit a much higher degree of spatial consistency, compared to previous automatic colour transfer methods. We also demonstrate that our method requires considerably less manual effort than previous user-assisted colorization methods Given a grayscale image to colorize, we first determine for each pixel which example segment it should learn it's colour from.

This is done automatically using a robust supervised classification scheme that analyses the low-level feature space defined by small neighbourhoods of pixels in the example image. Next, each pixel is assigned a colour from the appropriate region using a neighbourhood matching metric, combined with spatial filtering for improved spatial coherence. Each colour assignment is associated with a confidence value, and pixels with a sufficiently high confidence level are provided as "micro-scribbles" to the optimization-based colorization algorithm of Levin et al., which produces the final complete colorization of the image.

DESIGN AND IMPLEMENTATION:

Image colorization is the process of taking an **input grayscale (black and white) image** and then producing an **output colorized image** that represents the semantic colors and tones of the input (for example, an ocean on a clear sunny day must be plausibly "blue" — it can't be coloured "hot pink" by the model).

Previous methods for image colorization either:

1. Relied on significant human interaction and annotation
2. Produced desaturated colorization

The novel approach we are going to use here today instead relies on deep learning. We will utilize a Convolutional Neural Network capable of colorizing black and white images with results that can even "fool" humans!

Black and white image colorization with OpenCV and Deep Learning

In the first part of this tutorial, we'll discuss how deep learning can be utilized to colorize black and white images.

From there we'll utilize OpenCV to colorize black and white images for both:

1. Images

2. Video streams

We'll then explore some examples and demos of our work.

How can we colorize black and white images with deep learning?

The technique we'll be covering here today is from Zhang et al.'s 2016 ECCV paper, ***Colourful Image Colorization***.

Previous approaches to black and white image colorization relied on *manual human annotation* and often produced desaturated results that were not "believable" as true colorizations.

Zhang et al. decided to attack the problem of image colorization by using Convolutional Neural Networks to "hallucinate" what an input grayscale image would look like when colorized.

To train the network Zhang et al. started with the **ImageNet dataset** and converted all images from the RGB colour space to the **Lab colour space.**

Similar to the RGB colour space, the Lab colour space has *three channels*. But *unlike* the RGB colour space, Lab encodes colour information differently:

- The ***L* channel** encodes lightness intensity only
- The ***A* channel** encodes green-red.

- And the ***B* channel** encodes blue-yellow

A full review of the Lab colour space is outside the scope of this post (see **this guide** for more information on Lab), but the gist here is that Lab does a better job representing how humans see colour.

Since the *L* channel encodes only the intensity, **we can use the *L* channel as our grayscale input to the network.**

From there the network must **learn to predict the *a* and *b* channels.** Given the **input *L* channel** and the **predicted *ab* channels** we can then form our **final output image.**

The entire (simplified) process can be summarized as:

1. Convert all training images from the RGB colour space to the Lab colour space.
2. Use the ***L* channel** as the input to the network and train the network to predict the ***ab* channels.**
3. Combine the input ***L* channel** with the predicted ***ab* channels.**
4. Convert the Lab image back to RGB.

To produce more plausible black and white image colorizations the authors also utilize a few additional techniques including mean annealing and a specialized loss function for colour rebalancing (both of which are outside the scope of this post).

Colorizing black and white images with OpenCV

Let's go ahead and implement black and white image colorization script with OpenCV.

Open up the `colouriser.py` file and insert the following code:

```
colouriser.py > ...
1   import numpy as np
2   import cv2
3   import PySimpleGUI as sg
4   import os.path
5
6   version = '7 June 2020'
7
8   prototxt = r'model/colorization_deploy_v2.prototxt'
9   model = r'model/colorization_release_v2.caffemodel'
10  points = r'model/pts_in_hull.npy'
11  points = os.path.join(os.path.dirname(__file__), points)
12  prototxt = os.path.join(os.path.dirname(__file__), prototxt)
13  model = os.path.join(os.path.dirname(__file__), model)
14  if not os.path.isfile(model):
15      sg.popup_scrolled('Missing model file', 'You are missing the file "colorization_release_v2.caffemodel"
16                        'Download it and place into your "model" folder', 'You can download this file from t
17      exit()
18  net = cv2.dnn.readNetFromCaffe(prototxt, model)     # load model from disk
19  pts = np.load(points)
```

Our colorizer script only requires three imports: NumPy, OpenCV, and

`argparse`

.

Let's go ahead and **use `argparse` to parse command line arguments**. This script requires that these four arguments be passed to the script directly from the terminal:

- `--image`

  : The path to our input black/white image.

- `--prototxt`

  : Our path to the Caffe prototxt file.

- `--model`

  . Our path to the Caffe pre-trained model.

- `--points`

  : The path to a NumPy cluster center points file.

With the above four flags and corresponding arguments, the script will be able to run with different inputs without changing any code.

Let's go ahead and load our model and cluster centers into memory:

```
18  net = cv2.dnn.readNetFromCaffe(prototxt, model)     # load model from disk
19  pts = np.load(points)
20
21  # add the cluster centers as 1x1 convolutions to the model
22  class8 = net.getLayerId("class8_ab")
23  conv8 = net.getLayerId("conv8_313_rh")
24  pts = pts.transpose().reshape(2, 313, 1, 1)
25  net.getLayer(class8).blobs = [pts.astype("float32")]
26  net.getLayer(conv8).blobs = [np.full([1, 313], 2.606, dtype="float32")]
27
28  def colorize_image(image_filename=None, cv2_frame=None):
29      """
30      Where all the magic happens.  Colorizes the image provided. Can colorize either
31      a filename OR a cv2 frame (read from a web cam most likely)
32      :param image_filename: (str) full filename to colorize
33      :param cv2_frame: (cv2 frame)
34      :return: Tuple[cv2 frame, cv2 frame] both non-colorized and colorized images in cv2 format as a tuple
35      """
```

Line 21 loads our Caffe model directly from the command line argument values. OpenCV can read Caffe models via the

`cv2.dnn.readNetFromCaffe` function.

Line 22 then loads the cluster center points directly from the command line argument path to the points file. This file is in NumPy format so we're using

`np.load`.

From there, **Lines 25-29:**

- Load centers for ***ab* channel** quantization used for rebalancing.
- Treat each of the points as *1×1* convolutions and add them to the model.

Now let's load, scale, and convert our image:

```
36    # load the input image from disk, scale the pixel intensities to the range [0, 1], and then convert th
37    image = cv2.imread(image_filename) if image_filename else cv2_frame
38    scaled = image.astype("float32") / 255.0
39    lab = cv2.cvtColor(scaled, cv2.COLOR_BGR2LAB)
40
```

To load our input image from the file path, we use

`cv2.imread`
on **Line 34**.

Pre-processing steps include:

- Scaling pixel intensities to the range *[0, 1]* (**Line 35**).
- Converting from BGR to Lab colour space (**Line 36**).

Let's continue with our pre-processing:

```
41    # resize the Lab image to 224x224 (the dimensions the colorization network accepts), split channels, e
42    resized = cv2.resize(lab, (224, 224))
43    L = cv2.split(resized)[0]
44    L -= 50
45
```

We'll go ahead and resize the input image to *224×224* (**Line 41**), the required input dimensions for the network.

Then we grab the

`L`
channel only (i.e., the input) and perform mean subtraction (**Lines 42 and 43**).

Now we can pass the **input *L* channel** through the network to **predict the *ab* channels:**

```
46    # pass the L channel through the network which will *predict* the 'a' and 'b' channel values
47    'print("[INFO] colorizing image...")'
48    net.setInput(cv2.dnn.blobFromImage(L))
49    ab = net.forward()[0, :, :, :].transpose((1, 2, 0))
50
51    # resize the predicted 'ab' volume to the same dimensions as our input image
52    ab = cv2.resize(ab, (image.shape[1], image.shape[0]))
```

A forward pass of the `L` channel through the network takes place on **Lines 48 and 49** (here is a refresher on **OpenCV's blobFromImage** if you need it).

Notice that after we called `net.forward`, on the same line, we went ahead and extracted the predicted `ab` volume. I make it look easy here, but refer to the **Zhang et al. documentation and demo on GitHub** if you would like more details.

From there, we resize the predicted `ab` volume to be the same dimensions as our input image (**Line 53**). Now comes the time for post-processing. Stay with me here as we essentially go in reverse for some of our previous steps:

```
    # grab the 'L' channel from the *original* input image (not the resized one) and concatenate the origi
    L = cv2.split(lab)[0]
    colorized = np.concatenate((L[:, :, np.newaxis], ab), axis=2)

    # convert the output image from the Lab color space to RGB, then clip any values that fall outside the
    colorized = cv2.cvtColor(colorized, cv2.COLOR_LAB2BGR)
    colorized = np.clip(colorized, 0, 1)

    # the current colorized image is represented as a floating point data type in the range [0, 1] -- let'
    colorized = (255 * colorized).astype("uint8")
    return image, colorized


def convert_to_grayscale(frame):
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)  # Convert webcam frame to grayscale
    gray_3_channels = np.zeros_like(frame)  # Convert grayscale frame (single channel) to 3 channels
    gray_3_channels[:, :, 0] = gray
    gray_3_channels[:, :, 1] = gray
    gray_3_channels[:, :, 2] = gray
    return gray_3_channels

```

Post processing includes:

- Grabbing the `L` channel from the ***original*** input image (**Line 58**) and concatenating the original `L` channel and ***predicted*** `ab`channels together forming `colorized` (**Line 59**).
- Converting the `colorized` image from the Lab colour space to RGB (**Line 63**).
- Clipping any pixel intensities that fall outside the range *[0, 1]* (**Line 64**).
- Bringing the pixel intensities back into the range *[0, 255]* (**Line 69**). During the preprocessing steps (**Line 35**) we divided by `255` and now we are multiplying by `255`.
- I've also found that this scaling and `"uint8"` conversion isn't a requirement but that it helps the code work between **OpenCV 3.4.x** and **4.x** versions.

Finally, both our original `image` and `colorized` images are displayed on the screen!

Real-time black and white video colorization with OpenCV:

This script follows the same process as above except we'll be processing frames of a video stream. I'll be reviewing it in less detail and focusing on the frame grabbing + processing aspects.

Open up the webcam.py and insert the following code:

```
import numpy as np
import cv2
import PySimpleGUI as sg
import os.path

prototxt = r'model/colorization_deploy_v2.prototxt'
model = r'model/colorization_release_v2.caffemodel'
points = r'model/pts_in_hull.npy'
points = os.path.join(os.path.dirname(__file__), points)
prototxt = os.path.join(os.path.dirname(__file__), prototxt)
model = os.path.join(os.path.dirname(__file__), model)
if not os.path.isfile(model):
    sg.popup_scrolled('Missing model file', 'You are missing the file "colorization_release_v2.caffemodel"
                      'Download it and place into your "model" folder', 'You can download this file from t
    exit()
net = cv2.dnn.readNetFromCaffe(prototxt, model)     # load model from disk
pts = np.load(points)
```

Our video script requires two additional imports:

- VideoStream

  allows us to grab frames from a webcam or video file

- time

  will be used to pause to allow a webcam to warm up

Let's initialize our VideoStream now:

```
# initialize a boolean used to indicate if either a webcam or input
# video is being used
webcam = not args.get("input", False)
# if a video path was not supplied, grab a reference to the webcam
if webcam:
print("[INFO] starting video stream...")
vs = VideoStream(src=0).start()
time.sleep(2.0)
# otherwise, grab a reference to the video file
else:
print("[INFO] opening video file...")
vs = cv2.VideoCapture(args["input"])
```

Depending on whether we're working with a `webcam` or video file, we'll create our `vs` (i.e., "video stream") object here.

From there, we'll load the colorizer deep learning model and cluster centers (the same way we did in our previous script):

```
Black and white image colorization with OpenCV and Deep Learning
# load our serialized black and white colorizer model and cluster
# center points from disk
print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])
pts = np.load(args["points"])
# add the cluster centers as 1x1 convolutions to the model
class8 = net.getLayerId("class8_ab")
conv8 = net.getLayerId("conv8_313_rh")
pts = pts.transpose().reshape(2, 313, 1, 1)
net.getLayer(class8).blobs = [pts.astype("float32")]
net.getLayer(conv8).blobs = [np.full([1, 313], 2.606, dtype="float32")]
```

Now we'll start an infinite `while` loop over incoming frames. We'll process the frames directly in the loop:

```
Black and white image colorization with OpenCV and Deep Learning
# loop over frames from the video stream
while True:
# grab the next frame and handle if we are reading from either
# VideoCapture or VideoStream
frame = vs.read()
frame = frame if webcam else frame[1]
# if we are viewing a video and we did not grab a frame then we
```

```
# have reached the end of the video
if not webcam and frame is None:
break
# resize the input frame, scale the pixel intensities to the
# range [0, 1], and then convert the frame from the BGR to Lab
# color space
frame = imutils.resize(frame, width=args["width"])
scaled = frame.astype("float32") / 255.0
lab = cv2.cvtColor(scaled, cv2.COLOR_BGR2LAB)
# resize the Lab frame to 224x224 (the dimensions the colorization
# network accepts), split channels, extract the 'L' channel, and
# then perform mean centering
resized = cv2.resize(lab, (224, 224))
L = cv2.split(resized)[0]
L -= 50
```

Each frame from our `vs` is grabbed on **Lines 55 and 56**. A check is made for a

`None` type `frame` — when this occurs, we've reached the end of a video file (if we're processing a video file) and we can `break` from the loop (**Lines 60 and 61**).

Preprocessing (just as before) is conducted on **Lines 66-75**. This is where we resize, scale, and convert to Lab. Then we grab the `L` channel, and perform mean subtraction.

Let's now apply deep learning colorization and post-process the result:

```
Black and white image colorization with OpenCV and Deep Learning
# pass the L channel through the network which will *predict* the
# 'a' and 'b' channel values
net.setInput(cv2.dnn.blobFromImage(L))
ab = net.forward()[0, :, :, :].transpose((1, 2, 0))
# resize the predicted 'ab' volume to the same dimensions as our
# input frame, then grab the 'L' channel from the *original* input
# frame (not the resized one) and concatenate the original 'L'
# channel with the predicted 'ab' channels
ab = cv2.resize(ab, (frame.shape[1], frame.shape[0]))
L = cv2.split(lab)[0]
colorized = np.concatenate((L[:, :, np.newaxis], ab), axis=2)
# convert the output frame from the Lab color space to RGB, clip
```

```
# any values that fall outside the range [0, 1], and then convert
# to an 8-bit unsigned integer ([0, 255] range)
colorized = cv2.cvtColor(colorized, cv2.COLOR_LAB2BGR)
colorized = np.clip(colorized, 0, 1)
colorized = (255 * colorized).astype("uint8")
```

Our deep learning forward pass of `L` through the network results in the predicted `ab` channel.

Then we'll post-process the result to from our `colorized` image (**Lines 86-95**). This is where we resize, grab our ***original*** `L` , and concatenate our predicted `ab` . From there, we convert from Lab to RGB, clip, and scale.

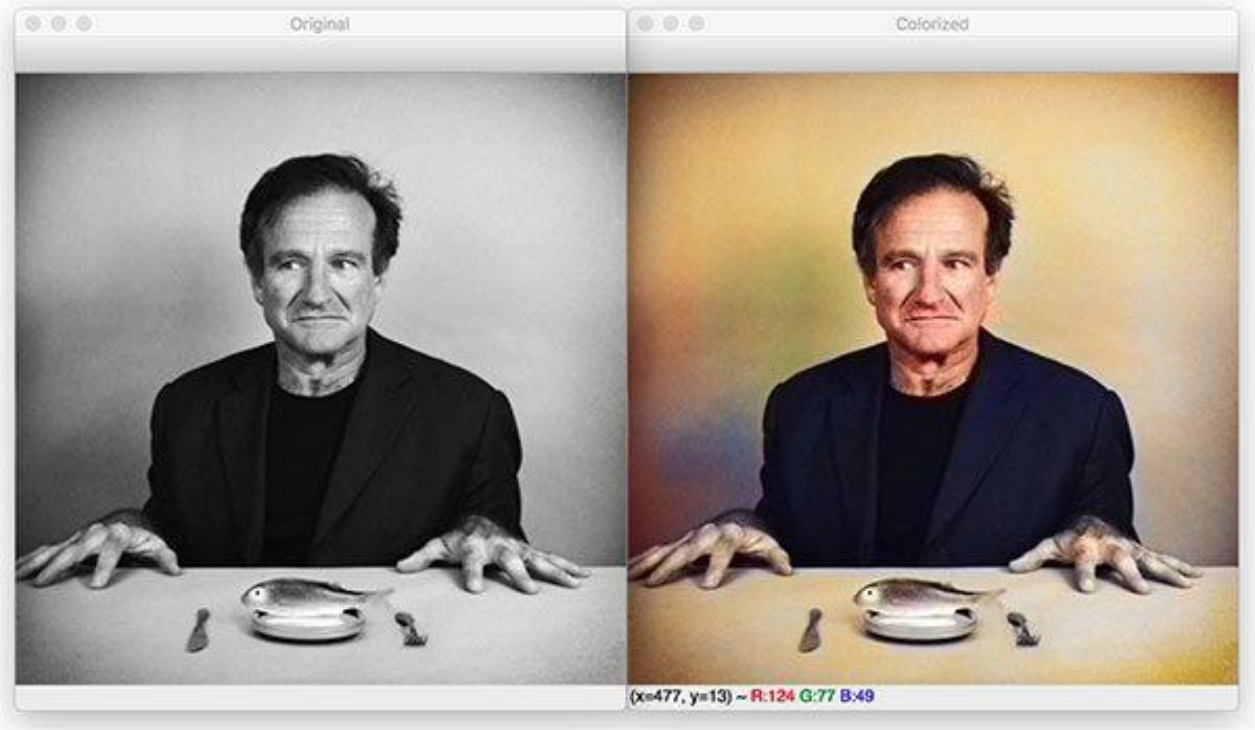
If you followed along closely above, you'll remember that all we do next is display the results:

```
Black and white image colorization with OpenCV and Deep Learning
# show the original and final colorized frames
cv2.imshow("Original", frame)
cv2.imshow("Grayscale", cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY))
cv2.imshow("Colorized", colorized)
key = cv2.waitKey(1) & 0xFF
# if the `q` key was pressed, break from the loop
if key == ord("q"):
break
# if we are using a webcam, stop the camera video stream
if webcam:
vs.stop()
# otherwise, release the video file pointer
else:
vs.release()
# close any open windows
cv2.destroyAllWindows()
```

Our original webcam `frame` is shown along with our grayscale image and `colorized` result.

Image colorization results:

Now that we've implemented our image colorization script, let's give it a try.



On the *left*, you can see the original input image of Robin Williams, a famous actor and comedian who passed away ~5 years ago.

On the *right*, you can see the output of the black and white colorization model.

CONCLUSION:

The image colorization model we used here was first introduced by Zhang et al. in their 2016 publication, *Colourful Image Colorization*.

Using this model, we were able to colorize both:

1. Black and white images
2. Black and white videos

Our results, while not perfect, demonstrated the plausibility of automatically colorizing black and white images and videos. According to Zhang et al., their approach was able to "fool" humans 32% of the time!

While image colorization is a boutique computer graphics task, it is also an instance of a difficult pixel prediction problem in computer vision.

Our method not only provides a useful graphics output, but can also be viewed as a pretext task for representation learning. Although only trained to colour, our network learns a representation that is surprisingly useful for object classification, detection, and segmentation, performing strongly compared to other self-supervised pre-training methods.

REFERENCES:

1. https://towardsdatascience.com/colorizing-old-b-w-photos-and-videos-with-the-help-of-ai-76ba086f15ec
2. https://pyimagesearch.com/2019/02/25/black-and-white-image-colorization-with-opencv-and-deep-learning/
3. https://richzhang.github.io/colorization/
4. https://www.rroij.com/open-access/coloring-of-black-and-white-imagesa-survey-38-40.pdf