Image Colorization with U-NET and GAN

*Deep Learning Group Project

Parth Kalkar
Dept. of Computer Science
Innopolis University
Innopolis, Russia
p.kalkar@innopolis.university

Hasan Khadra
Dept. of Computer Science
Innopolis University
Innopolis, Russia
h.khadra@innopolis.university

Umarov Doro
Dept. of Computer Science
Innopolis University
Innopolis, Russia
d.umarov@innopolis.ru

I. Introduction

One of the most exciting applications of deep learning is colorizing black and white images. This task needed a lot of human input and hardcoding several years ago, but now the process can be done end-to-end with AI and deep learning. For this task, you might need a lot of data or long training times to train your model from scratch. Still, in the last few weeks, we worked on this and tried many different model architectures, loss functions, training strategies, etc., and finally developed an efficient strategy to train such a model, using the latest advances in deep learning, on a rather small dataset and with short training times.

These days memories play an important role in our lives. All of the memories are captured either by pictures or videos. Everybody wants to relive the moments that took place in the past, but many of the memories are in black-white picture format.

Therefore we built a web app-based solution to relive those memories with color. This product will help you get a colored picture. As this product is AI-based, our main goal is to make it as accurate as possible.

This report will explain what we did to make this happen, including the code! and the strategies that helped and also those that were not useful. Before that, we will explain the colorization problem and briefly review what has been done in recent years.

II. Goals

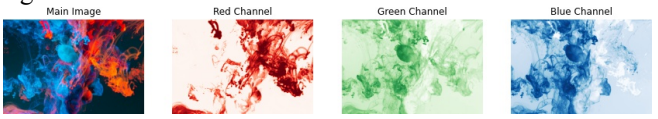
Build a fully functioning AI-based web app. The flexibility of this software solution covers all requirements, such as: delivering content, collecting evaluations, and easy deployment.

Try different Deep learning architectures/algorithms/models and choose the better one.

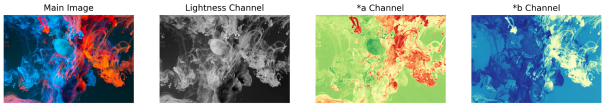
III. Introduction to the Colorization problem

A. RGB vs. L*a*b

As you might know, when we load an image, we get a rank-3 (height, width, color) array with the last axis containing the color data for our image. These data represent color in RGB color space, and 3 numbers for each pixel indicate how much Red, Green, and Blue the pixel is. In the following image, you can see that in the left part of the "main image" (the leftmost image), we have a blue color, so in the blue channel of the image, that part has higher values and has turned dark.



In L*a*b color space, we have three numbers again for each pixel, but these numbers have different meanings. The first number (channel), L, encodes the Lightness of each pixel, and when we visualize this channel (the second image in the row below), it appears as a black-and-white image. The *a and *b channels encode how much green-red and yellow-blue each pixel is, respectively. In the following image, you can see each channel of L*a*b color space separately.



In all papers, we studied and all codes we checked out on colorization on GitHub, people use L*a*b color space instead of RGB to train the models. There are a couple of reasons for this choice, but we'll give you an intuition of why we made this choice. To train a model for colorization, we should give it a grayscale image and hope that it will make it colorful. When using L*a*b, we can give the L channel to the model (which is the grayscale image) and want it to predict the other two channels (*a, *b), and after its prediction, we concatenate all the channels and we get our colorful image. But if you use RGB, you have first to convert your image to grayscale, feed the grayscale image to the model, and hope it will predict three numbers for you, which is a way more difficult and unstable task due to the many more possible combinations of 3 numbers compared to two numbers. If we assume we have 256 choices (in an 8-bit unsigned integer image, this is the real number of choices) for each number, predicting the three numbers for each of the pixels is choosing between 256^3 combinations which are more than 16 million choices. Still, when predicting two numbers, we have about 65000 choices (actually, we are not going to choose these numbers like a classification task wildly, and we just wrote these numbers to give you an intuition).

B. How to solve the problem

During the last few years, many different solutions have been proposed to colorize images using deep learning. The

colorful Image Colorization paper approached the problem as a classification task. They also considered the uncertainty of this problem (e.x. a car in the image can take on many different and valid colors, and we cannot be sure about any color for it); however, another paper approached the problem as a regression task (with some more tweaks!). Each approach has pros and cons, but we will use a different strategy in this project.

C. Strategy we used to solve the problem

Image-to-Image Translation with Conditional Adversarial Networks paper, which you may know by the name pix2pix, proposed a general solution to many image-to-image tasks in deep learning, one of which was colorization. In this approach, two losses are used: an L1 loss, which makes it a regression task, and an adversarial (GAN) loss, which helps to solve the problem in an unsupervised manner (by assigning the outputs a number indicating how "real" they look!).

In this project, we first implement what the authors did in the paper and then check if it is possible to introduce a whole new generator model and some tweaks in the training strategy, which significantly helps reduce the size of the needed dataset while getting amazing results.

IV. Solution

A. A deeper dive into GAN world

As mentioned earlier, we built a GAN (a conditional GAN, to be specific) and used an extra loss function, L1 loss. Let's start with the GAN.

As you might know, in a GAN, we have a generator and a discriminator model which learn to solve a problem together. In our setting, the generator model takes a grayscale (1-channel image) and produces a 2-channel image, a channel for *a and another for *b. The discriminator takes these two produced channels, concatenates them with the input grayscale image, and decides whether this new 3-channel image is fake or real. Of course, the discriminator also needs to see some real images (3-channel images again in Lab color space) that are not produced by the generator and should learn that they are real.

So what about the "condition" we mentioned? Well, that grayscale image that both the generator and discriminator see is the condition we provide to both models in our GAN, and we expect that they consider this condition.

Let's take a look at the math. Consider x as the grayscale image, z as the input noise for the generator, and y as the 2-channel output we want from the generator (it can also represent the two color channels of a real image). Also, G is the generator model, and D is the discriminator. Then the loss for our conditional GAN will be:

$$\mathcal{L}_{cGAN}(G,D) = \mathbb{E}_{x,y}[\log D(x,y)] + \mathbb{E}_{x,z}[\log(1 - D(x, G(x,z))]$$

Notice that x is given to both models, which is the condition we introduce to both players of this game. We will not feed an "n" dimensional vector of random noise to the generator as you might expect, but the noise is introduced as dropout layers in the generator architecture.

B. A deeper dive into GAN world

The earlier loss function helps to produce stunning colorful images that seem real, but to further help the models and introduce some supervision in our task, we combine this loss function with L1 Loss (you might know L1 loss as mean absolute error) of the predicted colors compared with the actual colors:

$$\mathcal{L}_{L1}(G) = \mathbb{E}_{x,y,z}[\|y - G(x,z)\|_1]$$

If we use L1 loss alone, the model still learns to colorize the images, but it will be conservative and most of the time uses colors like "gray" or "brown" because when it doubts which color is the best, it takes the average and uses these colors to reduce the L1 loss as much as possible (it is similar to the blurring effect of L1 or L2 loss in super-resolution task). Also, the L1 Loss is preferred over the L2 loss (or mean squared error) because it reduces the effect of producing grayish images. So, our combined loss function will be:

$$G^* = \arg\min_G \max_D \mathcal{L}_{cGAN}(G,D) + \lambda \mathcal{L}_{L1}(G)$$

where λ is a coefficient to balance the contribution of the two losses to the final loss (of course, the discriminator loss does not involve the L1 loss).

V. Dataset

A. COCO dataset (Heading 2)

COCO stands for Common Objects in Context, as the image dataset was created to advance image recognition. The COCO dataset contains challenging, high-quality visual datasets for computer vision, mostly state-of-the-art neural networks.

MS COCO (Microsoft Common Objects in Context) is a large-scale image dataset containing 328,000 images of everyday objects and humans. The dataset contains annotations you can use to train machine learning models to recognize, label, and describe objects.

B. Dataset Usage

The original paper uses the whole ImageNet dataset (with 1.3 million images!). Still, here we use only 8,000 images from the COCO dataset for training which we had available on my device. So our training set size is 0.6% of what was used in the paper!

The dataset is not a constraint as we can use almost any dataset for this task as long as it contains many different scenes and locations you hope it will learn to colorize. We can even use ImageNet, for example, but we will only need 8000 of its images for this project.

C. Dataset Loading

As we are opening this on Google Colab, we can uncomment and run the following to install fastai.

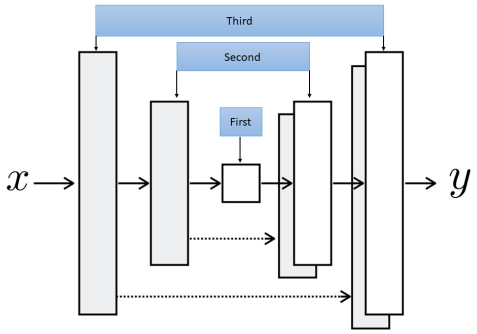
```
!pip install fastai==2.4
```

Almost all of the code in the notebook is with pure PyTorch. We need fastai here only to download part of the COCO dataset.

VI. Model

A. Generator Proposed by the paper.

This code implements a U-Net to be used as the generator of our GAN. The details of the code are out of the scope of this report, but the important thing to understand is that it makes the U-Net from the middle part of it (down in the U shape) and adds down-sampling and up-sampling modules to the left and right of that middle module (respectively) at every iteration until it reaches the input module and output module. Look at the following image that we made from one of the images in the article to give you a better sense of what is happening in the code:



The blue rectangles show the order in which the related modules are built with the code. The U-Net we will build has more layers than depicted in this image, but it suffices to give you the idea. Also, notice in the code that we are going eight layers down, so if we start with a 256 by 256 image in the middle of the U-Net, we will get a 1 by 1 ($256 / 2^8$) image, and then it gets up-sampled to produce a 256 by 256 image (with two channels). This code snippet is fascinating, and we highly recommend playing with it to grasp what every line of it is doing fully.

B. Discriminator

The architecture of our discriminator is rather straightforward. This code implements a model by stacking blocks of Conv-BatchNorm-LeackyReLU to decide whether the input image is fake or real. Notice that the first and last blocks do not use normalization and the last block has no activation function. Let's look at its blocks:

```
PatchDiscriminator(3)

PatchDiscriminator(
  (model): Sequential(
    (0): Sequential(
      (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (2): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
    (4): Sequential(
      (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))
    )
  )
)
```

We use a "Patch" Discriminator. Okay, what is it?! In a vanilla discriminator, the model outputs one number (a scaler), which represents how much the model thinks the input (which is the whole image) is real (or fake). In a patch discriminator, the model outputs one number for every patch of, say, 70 by 70 pixels of the input image, and each of them decides whether it is fake or not separately. Using such a model for the task of colorization seems reasonable because the local changes that the model needs to make are important, and maybe deciding on the whole image as in vanilla discriminator cannot take care of the subtleties of this task. Here, the model's output shape is 30 by 30, but it does not mean that our patches are 30 by 30. The actual patch size is obtained when you compute the receptive field of each of these 900 (30 multiplied by 30) output numbers, which in our case will be 70 by 70.

C. GAN Loss

This is a handy class we can use to calculate the GAN loss of our final model. In the init, we decide which kind of loss we're going to use (which will be "vanilla" in our project) and register some constant tensors as the "real" and "fake" labels. Then when we call this module, it makes an appropriate tensor full of zeros or ones (according to what we need at the stage) and computes the loss.

D. Model Initialization

We will initialize the weights of our model with a mean of 0.0 and a standard deviation of 0.02, which are the proposed hyperparameters.

E. Training

In the *init,* we define our generator and discriminator using the previous functions and classes we defined, and we also initialize them with the init_model function. Then we define our two loss functions and the optimizers of the generator and discriminator.

The whole work is being done in the optimization method of this class. First and only once per iteration (batch of the training set), we call the module's forward method and store the outputs in the fake_color variable of the class.

Then, we first train the discriminator by using the *backward_D* method in which we feed the fake images produced by the generator to the discriminator (make sure to detach them from the generator's graph so that they act as a constant to the discriminator, like normal images) and label them as fake.

Then we feed a batch of real images from training sets to the discriminator and label them as real. We add up the two losses for fake and real, take the average, and then call the backward on the final loss. Now, we can train the generator. In the backward_G method, we feed the discriminator the

Fake image and try to fool it by assigning real labels to them and calculating the adversarial loss.

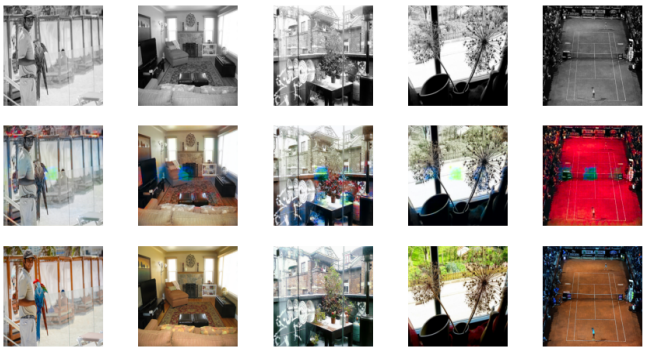
As mentioned earlier, we use L1 loss as well and compute the distance between the predicted two channels and the target two channels and multiply this loss by a coefficient (which is 100 in our case) to balance the two losses and then add this to the adversarial loss. Then we call the backward method of the loss.

Every epoch takes about 4 minutes on not a GPU as powerful as the Nvidia P5000. So if you are using 1080Ti or higher, it will be much faster.

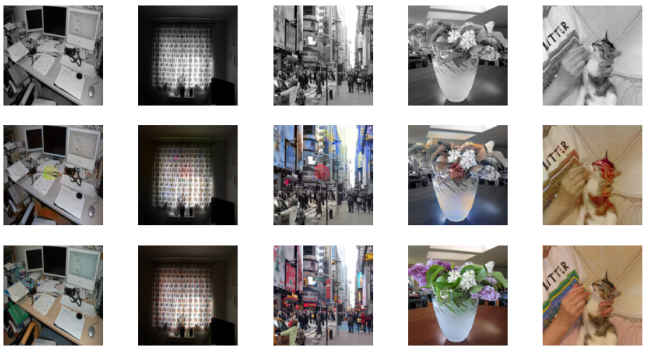
F. Results

Epoch	loss_D_fake	Loss_D	loss_D_real	loss_G_GAN:	loss_G_L1	loss_G
5	0.52271	0.55515	0.58760	1.24612	10.92885	12.17497
10	0.51748	0.54679	0.57610	1.26481	10.57909	11.84390
15	0.50693	0.53339	0.55985	1.26905	9.99937	11.26843
20	0.50832	0.52814	0.54796	1.30233	9.38216	10.68449
25	0.52603	0.54133	0.55662	1.30400	8.75683	10.06083
30	0.52737	0.53602	0.54468	1.29681	8.17807	9.47488
35	0.55535	0.55738	0.55940	1.31022	7.70705	9.01727
40	0.53682	0.54192	0.54703	1.31163	7.39645	8.70808

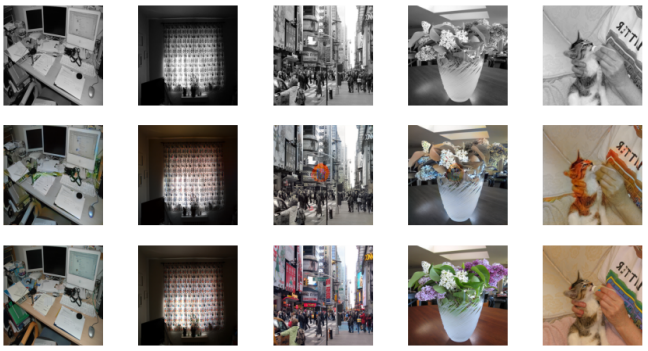
10th Epoch



20th Epoch



30th Epoch



40th Epoch

Every epoch takes about 8 to 9 minutes on Colab. After about 20 epochs, you should see some good results.

We let the model train for some longer (about 100 epochs). Here are the results of our baseline model:

G. Conclusion

As you can see, although this baseline model has some basic understanding of some most common objects in images like the sky and trees, … its output is far from something appealing. It cannot decide on the color of rare objects. It also displays some color spillovers and a circle-shaped mass of color (center of the first image of the second row) which is not good. So, with this small dataset, we cannot get good results with this strategy.

H. Model Files

Image_Colorization_model

VII. Web-App

A. Frontend

We developed a web application that uses our model to take gray images as input and produce the colored image based on that input. We used React.js and Typescript on the frontend side and Flask and Pytorch on the backend side. The flow of the application goes as follows: The user uploads a gray image to the website. The front end requests the backend to colorize the image using our model. The backend loads our pre-trained model Generator, feeds it with the gray image, generates the colored image, and sends it back to the frontend side.

To run the application, you'll need some confidential environment variables. You can contact us to set up your environment so you can use the web app locally.

B. Backend

To process the images from web applications, a backend was developed. The backend was written in Python using the Flask framework for route handling.

In the first stage, the backend loads the environment variables and the model, which can take up to a few seconds. After initialization, the backend will wait for the main route with POST and GET requests. On POST request, we obtain the image from the handler and check if the file format is correct before pushing it into the next stage. After the image was checked for file format, we moved it into our model to generate the colors. This process can take up to a few seconds as well. When the processing has been finished, we create the image by first saving it into our folder of images and then sending the saved image to the front end, where it will be displayed.

VIII. GitHub

Code

IX. Demo

Link

X. References

1. Zhang, Richard, Phillip Isola, and Alexei A. Efros. "Colorful image colorization." European conference on computer vision. Springer, Cham, 2016.
2. Isola, Phillip, et al. "Image-to-image translation with conditional adversarial networks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.