Colorization using CNN

Bereket Eshete . Prof Changdong Yoo

June 2019 KAIST

Abstract

This is a final project for spring 2019, EE331 Introduction to Machine Learning. Our goal is to perform colorization from given input images. In this report, we will construct a machine-learning model to automatically turn grayscale images into colored images. We will build the model using python programming language and PyTorch, we will review the basic tools and techniques we need for this task step by step. One unique essential part of the project is the normal images are composed of RGB values; however, we will represent these images into LAB values for our mission.

Keywords: Colorization, CNN

Purpose: Create Algorithm that returns identical size of color image from gray scale image.

Introduction

In image colorization, our goal is to produce a colored image given a grayscale input image. This issue is challenging on the grounds that a solitary grayscale picture may compare to numerous conceivable hued pictures. Thus, customary models regularly depended on user contribution in conjunction with a grayscale picture. As of late, profound neural systems have indicated exceptional achievement in programmed picture colorization going from grayscale to color with no extra human feedback. This achievement may to some degree be because of their capacity to catch and utilize semantic data in colorization; however, we are not yet sure, what exactly makes these types of models perform so well. Before clarifying the model, we will primarily spread out our problem to be solved accurately.

The Problem

Our goal is to deduce a full-colored image, which has three values per pixel (lightness, saturation, and hue), from a grayscale image, which has only one value per pixel (lightness only). We will work with images of size 256 x 256, hence our inputs are of size 256 x 256 x1(the lightness channel) and our outputs are of size 256 x 256 x2 (the other two channels). Instead of images in the RGB format, we will work with them in the LAB colorspace (Lightness, A, and B). This colorspace contains exactly the same information as RGB, but it will make it simpler for us to separate out the lightness channel from the other two. We will try to predict the color values of the input image directly through regression.

The Data

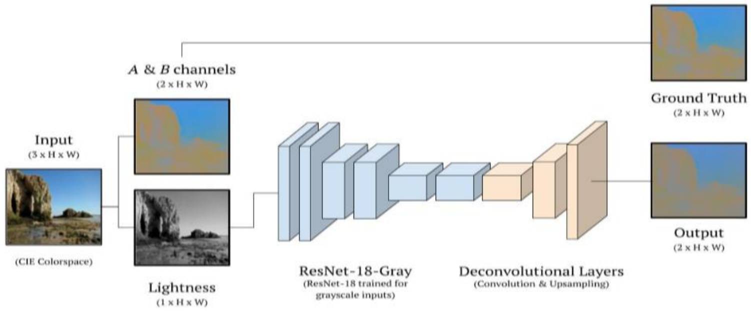
Data for colorization can be found everywhere and is easily accessible for anyone with internet; we can extract the grayscale channel from any colored image. For this project, we will use a subset of the KAIST places dataset of places, landscapes, and buildings provided by instructors. The full data can be found using the link below.

https://drive.google.com/drive/folders/1RFDx_CJSfQ0phm16Lw04wRDd9wbU41Qc?usp=sharing. Supplementary description of the data and performance results using this data by other students can also be found here. https://competitions.codalab.org/competitions/23046. The data comprises of training data and validation data. I have modified the initial directory for simplicity from “train_valid/trainset” to “train_valid/trainset/class” similar to the subfolder valset.

Train data directory [36,527 images]: “train_valid/trainset/class”
Validation data directory [36,527 images]: “train_valid/valset/class”

The Model

The model we will use is a CNN (convolutional neural network) we learned in this typical network as part of neural networks. We initially apply convolutional layers to extract features from the image, and then we apply deconvolutional layers to increase the spacial resolution of our features. This model begins with ResNet-18, an image classification network with 18 layers and residual connections (look the figure below). The first layer of the network accepts grayscale instead of colored input and is cut after the 6th set of layers.



We refer L as lightness. We cannot know the color of the image without the A and B values. If we know only the L value of the image, let's say that the image is L image.
Our detail task is as follows:

1. Given LAB train images, we have to construct the model and train it.
   - The input of the model should be L images only with lightness.
   - The output of the model can be A, and B values or LAB images
2. Given L validation images, we have to reconstruct the LAB images using the trained model.

We will define our model in python starting with the upsampling layers.

```
class ColorizationNet(nn.Module):
  def __init__(self, input_size=128):
    super(ColorizationNet, self).__init__()
    MIDLEVEL_FEATURE_SIZE = 128

    ## First half: ResNet
    resnet = models.resnet18(num_classes=365)
    # Change first conv layer to accept single-channel (grayscale) input
    resnet.conv1.weight = nn.Parameter(resnet.conv1.weight.sum(dim=1).unsqueeze(1))
    # Extract midlevel features from ResNet-gray
    self.midlevel_resnet = nn.Sequential(*list(resnet.children())[0:6])

    ## Second half: Upsampling
    self.upsample = nn.Sequential(
      nn.Conv2d(MIDLEVEL_FEATURE_SIZE, 128, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(128),
      nn.ReLU(),
      nn.Upsample(scale_factor=2),
      nn.Conv2d(128, 64, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(64),
      nn.ReLU(),
      nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(64),
      nn.ReLU(),
      nn.Upsample(scale_factor=2),
      nn.Conv2d(64, 32, kernel_size=3, stride=1, padding=1),
      nn.BatchNorm2d(32),
      nn.ReLU(),
      nn.Conv2d(32, 2, kernel_size=3, stride=1, padding=1),
      nn.Upsample(scale_factor=2)
    )

  def forward(self, input):

    # Pass input through ResNet-gray to extract features
    midlevel_features = self.midlevel_resnet(input)

    # Upsample to get colors
    output = self.upsample(midlevel_features)
    return output
```

After creating our model, we load it onto the GPU.

```
    midlevel_features = self.midlevel_resnet(input)

    # Upsample to get colors
    output = self.upsample(midlevel_features)
    return output

model = ColorizationNet()
```

Training

► Loss function

For loss function, we will use a mean squared error loss function since we are performing regression. We minimize the squared distance between the color value we try to predict, and the ground-truth color value.

```
model = ColorizationNet()

criterion = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=1e-2, weight_decay=0.0)
```

This misfortune capacity is somewhat problematic for colorization due to the multi-modality of the issue. For instance, if a dark dress could be red or blue, and our model picks the wrong color, it will be cruelly punished. Subsequently, our model will for the most part pick desaturated hues that are more averse to be "wrong" than splendid, dynamic hues.

► Optimizer

We optimize the loss function with the Adam optimizer.

```
model = ColorizationNet()

criterion = nn.MSELoss()

optimizer = torch.optim.Adam(model.parameters(), lr=1e-2, weight_decay=0.0)
```

► Loading data

We use torch text to load the data. Since we need pictures in the LAB space, we initially need to characterize a custom information loader to change over the pictures.

```
class GrayscaleImageFolder(datasets.ImageFolder):
  '''Custom images folder, which converts images to grayscale before loading'''
  def __getitem__(self, index):
    path, target = self.imgs[index]
    img = self.loader(path)
    if self.transform is not None:
      img_original = self.transform(img)
      img_original = np.asarray(img_original)
      img_lab = rgb2lab(img_original)
      img_lab = (img_lab + 128) / 255
      img_ab = img_lab[:, :, 1:3]
      img_ab = torch.from_numpy(img_ab.transpose((2, 0, 1))).float()
      img_original = rgb2gray(img_original)
      img_original = torch.from_numpy(img_original).unsqueeze(0).float()
    if self.target_transform is not None:
      target = self.target_transform(target)
    return img_original, img_ab, target
```

Next, we define transforms for the training and validation data.

```
# Training
train_transforms = transforms.Compose([transforms.RandomResizedCrop(224), transforms.RandomHorizontalFlip()])
train_imagefolder = GrayscaleImageFolder('train_valid/trainset', train_transforms)
train_loader = torch.utils.data.DataLoader(train_imagefolder, batch_size=64, shuffle=True)

# Validation
val_transforms = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224)])
val_imagefolder = GrayscaleImageFolder('train_valid/valset' , val_transforms)
val_loader = torch.utils.data.DataLoader(val_imagefolder, batch_size=64, shuffle=False)
```

► Supplementary functions

Before we train, we characterize supplementary functions for tracking the training loss and converting images back to RGB.

```
class AverageMeter(object):
  '''A handy class from the PyTorch ImageNet tutorial'''
  def __init__(self):
    self.reset()
  def reset(self):
    self.val, self.avg, self.sum, self.count = 0, 0, 0, 0
  def update(self, val, n=1):
    self.val = val
    self.sum += val * n
    self.count += n
    self.avg = self.sum / self.count

def to_rgb(grayscale_input, ab_input, save_path=None, save_name=None):
  '''Show/save rgb image from grayscale and ab channels
     Input save_path in the form {'grayscale': '/path/', 'colorized': '/path/'}'''
  plt.clf() # clear matplotlib
  color_image = torch.cat((grayscale_input, ab_input), 0).numpy() # combine channels
  color_image = color_image.transpose((1, 2, 0))  # rescale for matplotlib
  color_image[:, :, 0:1] = color_image[:, :, 0:1] * 100
  color_image[:, :, 1:3] = color_image[:, :, 1:3] * 255 - 128
  color_image = lab2rgb(color_image.astype(np.float64))
  grayscale_input = grayscale_input.squeeze().numpy()
  if save_path is not None and save_name is not None:
    plt.imsave(arr=grayscale_input, fname='{}{}'.format(save_path['grayscale'], save_name), cmap='gray')
    plt.imsave(arr=color_image, fname='{}{}'.format(save_path['colorized'], save_name))
```

► Validation

During validation, we run the model without backpropagation using torch.no_grad.

```python
def validate(val_loader, model, criterion, save_images, epoch):
  model.eval()

  # Prepare value counters and timers
  batch_time, data_time, losses = AverageMeter(), AverageMeter(), AverageMeter()

  end = time.time()
  already_saved_images = False
  for i, (input_gray, input_ab, target) in enumerate(val_loader):
    data_time.update(time.time() - end)

    # Use GPU
    if use_gpu: input_gray, input_ab, target = input_gray.cuda(), input_ab.cuda(), target.cuda()

    # Run model and record loss
    output_ab = model(input_gray) # throw away class predictions
    loss = criterion(output_ab, input_ab)
    losses.update(loss.item(), input_gray.size(0))

    # Save images to file
    if save_images and not already_saved_images:
      already_saved_images = True
      for j in range(min(len(output_ab), 10)): # save at most 5 images
        save_path = {'grayscale': 'outputs/gray/', 'colorized': 'outputs/color/'}
        save_name = 'img-{}-epoch-{}.jpg'.format(i * val_loader.batch_size + j, epoch)
        to_rgb(input_gray[j].cpu(), ab_input=output_ab[j].detach().cpu(), save_path=save_path, save_name=save_name)

    # Record time to do forward passes and save images
    batch_time.update(time.time() - end)
    end = time.time()

    # Print model accuracy -- in the code below, val refers to both value and validation
    if i % 25 == 0:
      print('Validate: [{0}/{1}]\t'
            'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
            'Loss {loss.val:.4f} ({loss.avg:.4f})\t'.format(
             i, len(val_loader), batch_time=batch_time, loss=losses))

  print('Finished validation.')
  return losses.avg
```

► Training

During training, we execute the model and backpropagate using loss.backward(). We intially define a function that trains for just one epoch.

```python
def train(train_loader, model, criterion, optimizer, epoch):
  print('Starting training epoch {}'.format(epoch))
  model.train()

  # Prepare value counters and timers
  batch_time, data_time, losses = AverageMeter(), AverageMeter(), AverageMeter()

  end = time.time()
  for i, (input_gray, input_ab, target) in enumerate(train_loader):

    # Use GPU if available
    if use_gpu: input_gray, input_ab, target = input_gray.cuda(), input_ab.cuda(), target.cuda()

    # Record time to load data (above)
    data_time.update(time.time() - end)

    # Run forward pass
    output_ab = model(input_gray)
    loss = criterion(output_ab, input_ab)
    losses.update(loss.item(), input_gray.size(0))

    # Compute gradient and optimize
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Record time to do forward and backward passes
    batch_time.update(time.time() - end)
    end = time.time()

    # Print model accuracy -- in the code below, val refers to value, not validation
    if i % 25 == 0:
      print('Epoch: [{0}][{1}/{2}]\t'
            'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
            'Data {data_time.val:.3f} ({data_time.avg:.3f})\t'
            'Loss {loss.val:.4f} ({loss.avg:.4f})\t'.format(
          epoch, i, len(train_loader), batch_time=batch_time,
          data_time=data_time, loss=losses))

  print('Finished training epoch {}'.format(epoch))
```

Next, we define a loop to train for 100 epochs:

```python
# Train model
for epoch in range(epochs):
  # Train for one epoch, then validate
  train(train_loader, model, criterion, optimizer, epoch)
  with torch.no_grad():
    losses = validate(val_loader, model, criterion, save_images, epoch)
  # Save checkpoint and replace old best model if current model is better
  if losses < best_losses:
    best_losses = losses
    torch.save(model.state_dict(), 'checkpoints/model-epoch-{}-losses-{:.3f}.pth'.format(epoch+1,losses))
```

Results and Analysis

The result of the project are colorized images. The parameters we want to consider to measure the performance of this algorithm is the loss. Hence, we will see how the loss function responds to different data size and epoch.

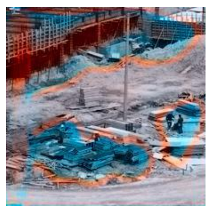
We can execute the python file by typing “example.py” or for this report “colorize.py”

- For data= 1, epoch =100, Loss = 0.0058

```
C:\2019\Spring 2019\Introduction to Machine Learning\Project\Example>py example.py
Starting training epoch 0
Epoch: [0][0/1] Time 5.537 (5.537)      Data 1.801 (1.801)      Loss 0.2496 (0.2496)
Finished training epoch 0
Validate: [0/1] Time 6.056 (6.056)      Loss 0.1088 (0.1088)
Finished validation.
Starting training epoch 1
Epoch: [1][0/1] Time 0.991 (0.991)      Data 0.098 (0.098)      Loss 0.0662 (0.0662)
Finished training epoch 1
C:\Users\aspire\Miniconda3\lib\site-packages\skimage\color\colorconv.py:988: UserWarning: Color data out of range: Z < 0 in 49675 pixels
  warn('Color data out of range: Z < 0 in %s pixels' % invalid[0].size)
Validate: [0/1] Time 0.623 (0.623)      Loss 32.9370 (32.9370)
Finished validation.
```

going in loop from epoch 1 to 100 ..

```
Starting training epoch 99
Epoch: [99][0/1]        Time 0.971 (0.971)      Data 0.050 (0.050)      Loss 0.0002 (0.0002)
Finished training epoch 99
C:\Users\aspire\Miniconda3\lib\site-packages\skimage\color\colorconv.py:988: UserWarning: Col
  warn('Color data out of range: Z < 0 in %s pixels' % invalid[0].size)
Validate: [0/1] Time 0.514 (0.514)      Loss 0.0058 (0.0058)
Finished validation.
```

The image on the right is colorized from the grey left image

- For data= 5, epoch =100, Loss = 0.0019

```
Starting training epoch 99
Epoch: [99][0/1]        Time 4.318 (4.318)      Data 0.243 (0.243)      Loss 0.0004 (0.0004)
Finished training epoch 99
Validate: [0/1] Time 2.066 (2.066)      Loss 0.0019 (0.0019)
Finished validation.
```

The image on the right is colorized from the grey left image

- For data = 10, For data= 5, epoch =100, Loss = 0.0016

```
Starting training epoch 99
Epoch: [99][0/1]        Time 6.048 (6.048)      Data 0.407 (0.407)      Loss 0.0007 (0.0007)
Finished training epoch 99
Validate: [0/1] Time 3.192 (3.192)      Loss 0.0016 (0.0016)
Finished validation.
```

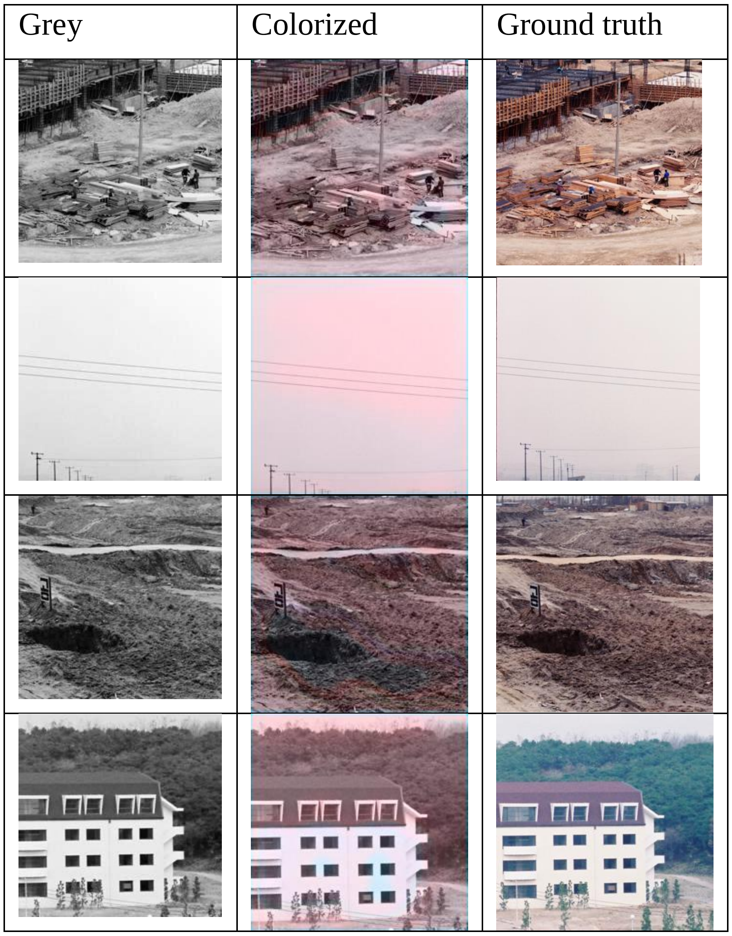


Fig 1. Comparing input and result, for this project. Looks impressive since I used a max of 10 data images due to a limit performance of my computer, however with good computer, one can use all 36,527 images as training and gain better outstanding result.

Conclusion

In this report, we built a simple automatic image colorizer using PyTorch by utilizing CNN and showed it is possible to perform automatic colorization. From the result we obtained this research area we can observe that this areas has a great future, especially in converting old white and black videos to color-coded videos.

References

[1]https://drive.google.com/drive/folders/1RFDx_CJSfQ0phm16Lw04wRDd9wbU41Qc?usp=sharing.
[2] https://competitions.codalab.org/competitions/23046
[3] https://richzhang.github.io/ideepcolor/
[4]” https://lukemelas.github.io/image-colorization.html”