RUTGERS

THE STATE UNIVERSITY
OF NEW JERSEY

CS520 - INTRO TO ARTIF INTEL

Assignment 4 Colorizer

Team member:
Xinyu Lyu(xl422)
Weijia Sun(ws368)
Xun Tang (xt63)
Yi Wu (yw641)

Contributions All Equal !!!

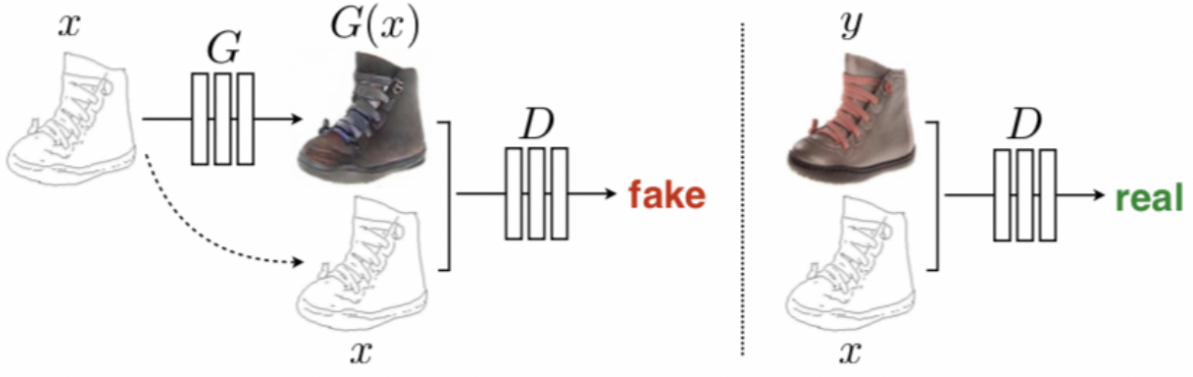
Representing the Process

How can you represent the coloring process in a way that a computer can handle? What spaces are you mapping between? What maps do you want to consider? Note that mapping from a single grayscale value gray to a corresponding color (r, g, b) on a pixel by pixel basis, you do not have enough information in a single gray value to reconstruct the correct color (usually).

We represent out coloring process by using the deep neural net architectures called Generative adversarial networks (GAN) which consists with two nets, pitting one against the other. For both the generator and the discriminator, we use the Convolutional Neural Network to construct them. Since the colorized image is generated by the CNN in the generator, the mapping process from the grayscale image to a colorful image is done by the CNN model. And as we use c-GAN as models, when training the generator G we both need the real colorful image as its training output target and its grayscale image as the conditional input image. And for the discriminator D, we both need the real colorful image and its grayscale image input image to train it. So, we use this concatenated image as training input in each batch of training which is more convenient for our model training. Therefore, the cGAN is a supervised neural network learning model.

Methods for GAN

Generally, GAN learns to map from random noise vector z to output image y, G: z->y [1]. In contrast, conditional GANs learns to map from observed image and random noise vector z to y, G{x, z}->y. Both in GAN and c-GAN, the generator G learns to generate a ‘real’ image based on the conditional image which cannot be distinguished as fake by an adversarial trained discriminator D. And the discriminator D learns to detect if the image, generated by the generator, is fake or not based on the conditional image. And the process of cGAN has been shown as the figure shown below.



The training target can be expressed as follows,

$$L_{cGAN}(G, D) = E_{x,y}[logD(x, y)] + E_{x,y}[log(1 - D(x, G(x, z))] \quad (1)$$

G tries to minimize the loss function by increase D(x, G(x, z)), generating a 'real' image to fool the discriminator. Meanwhile, D tries to maximize it against an adversarial G. So, in such a mutual restraint learning process, G tries to generate a more 'real' image, and D attempts to distinguish the image is fake or not. And the former is just what we want. i.e.

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

Network Architecture

Generator

In the GAN, we have the generator which is responsible for generating the "fake colorful" image from the input grayscale image. The generator generates a CNN model which accept the input of the greyscale pictures and output the corresponding colorized image.

Downsampling layer

The CNN model for generator mainly has two parts. The first is downsampling layers and the second is an upsampling layer. Also, for the beginning and the end layer is the input and the output. For downsampling layers, we designed a 7-layer architecture, whose the number of filters increases from 2 to 8. The reason we bring the downsampling layer in is that: intuitively we actually try to let the CNN focus on the fewer features than all of it, mostly because to reduce the redundancy in the feature map which helps in a faster time and lower memory while training. Through the downsampling layer, we can reduce the dimension of the feature map, ignore the relative position of the image and avoiding the overfitting to some extent.

We encapsulate the logic code for generating the downsampling layer of the CNN in the method: `conv2d`.

```
def conv2d(layer_input, filters, f_size=4, bn=True):
    """Layers used during downsampling"""
    d = Conv2D(filters, kernel_size=f_size, strides=2, padding='same')(layer_input)
    d = LeakyReLU(alpha=0.2)(d)
    if bn:
        d = BatchNormalization(momentum=0.8)(d)
    return d
```

We use the Conv2D from Keras. This method creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. For the arguments here:

- **filters**: Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).

- **kernel_size**: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides**: An integer, specifying the strides of the convolution along the height and width. Also, it can specify the same value for all spatial dimensions.
- **padding**: "same" results in padding the input such that the output has the same length as the original input.

For what is the batch normalization layer and why do we use it here, you may go to the supplementary material for more information.

Upsampling layer

After the downsampling, we designed upsampling layers to accept the result from layers before to reconstruct the image. Also, in the last layer, we use a convolution layer to sharp the image and make the result looks better.
We encapsulate the logic code for generating the upsampling layer of the CNN in the method: `deconv2d`.

```
    def deconv2d(layer_input, skip_input, filters, f_size=4, dropout_rate=0):
        """Layers used during upsampling"""
        u = UpSampling2D(size=2)(layer_input)
                    u = Conv2D(filters, kernel_size=f_size, strides=1, padding='same',
activation='relu')(u)
        u = BatchNormalization(momentum=0.8)(u)
        u = Concatenate()([u, skip_input])
        return u
```

This is the upsampling layer for 2D inputs.

The method UpSampling2D repeats the rows and columns of the data by size[0] and size[1] respectively. The argument size means the upsampling factors for rows and columns.

Then we use the Conv2D again to finished generating the upsampling layer for our CNN model. For the upsampling layer, we use the ReLU as the activation function. For what is the ReLU/LeakyReLU and why do we use the ReLU/LeakyRelu here, you may go to the supplementary material for more information.

For the design of the upsampling layers, we bring the mechanism of U-Net in. By using the U-Net we can let some specific features shuttle directly across the net in order to let the input and the output layer share some low-level information(features).

Discriminator

The discriminator generates a CNN model, which accepts two images. One is the origin grayscale image and the other is the colorized image you want the discriminator to tell

whether it is the real colorized image. The model's output is the validity which represents how likely the colorized image is the real one. The validity range from [0, 1].

For the discriminator, we use similar network architecture with the generator.

We encapsulate the logic code for generating the layer of the CNN in the method: d_layer.

```
def d_layer(layer_input, filters, f_size=4, bn=True):
    """Discriminator layer"""
    d = Conv2D(filters, kernel_size=f_size, strides=2, padding='same')(layer_input)
    d = LeakyReLU(alpha=0.2)(d)
    if bn:
        d = BatchNormalization(momentum=0.8)(d)
    return d
```

For the optimizer of the discriminator, we choose the Stochastic gradient descent (often shortened to SGD), also known as incremental gradient descent, is an iterative method for optimizing a differentiable objective function, a stochastic approximation of gradient descent optimization. It is called stochastic because samples are selected randomly (or shuffled) instead of as a single group (as in standard gradient descent) or in the order they appear in the training set.

$$w := w - \eta \nabla Q_i(w).$$

Keras SGD: learning rate 0.01. Qi; MSE(discriminator)

For the CNN model, the first layer takes the combined image as the input. Then we designed three layers for downsampling to extract the feature. We use the convolution layer as the last one to determine how likely this is the fake image.

Markovian discriminator (PatchGAN)

As we know, using L1 loss will lead to blurry results on image generation, which is enough for the low-resolution request. However, we want to make our model also sufficient for high-resolution images. Therefore, we use PatchGAN structures that paying attention to local images patches. We split the image into small patches and tell if the image is fake or real in each patch.

```
valid = np.ones((batch_size,) + self.disc_patch)
fake = np.zeros((batch_size,) + self.disc_patch)
```

Together with the codes, we generate two arrays valid and zeros as the target for real image and the fake image in each N*N patch, whose size is (batch_size,16,16,1). During experiments, we know that the 16*16 PatchGAN is enough to promote sharp outputs as

high-quality results. And such small size has fewer parameters and runs faster, so we choose to use 16*16 PatchGAN to represent the validity of each image.

Data

Where are you getting your data from to train/build your model? What kind of pre-processing might you consider doing?

We use python crawler to download 125 photos with beaches from google pictures. Then, we split it into train dataset and test dataset with the scale as 4:1.

For both the train dataset/test dataset pictures, we first scale up the pictures into 256*256 size with the help from the extra library PIL. Then, we convert them into grayscale images with the support from the extra library PIL. At last, we concatenate the colorful image together with its corresponding grayscale image with the size of 512*256 shown below.

That is because our model only takes 256*256 as input image's size and we use c-GAN as models when training the generator G we both need the real colorful image as its training output target and its grayscale image as the conditional input image. And for the discriminator D, we both need the real colorful image and its grayscale image input image to train it. So, we use this concatenated image as training input in each batch of training which is more convenient for our model training.

Evaluating the Model

Given a model for moving from grayscale images to color images (whatever spaces you are mapping between), how can you evaluate how good your model is? How can you assess the error of your model (hopefully in a way that can be learned from)? Note there are at least two things to consider when thinking about the error in this situation: numerical/quantified error (in terms of deviation between predicted and actual) and perceptual error (how good do humans find the result of your program).

Numerical Error

For discriminator, we choose the MSE (mean squared error) and for the generator, we choose the MAE (Mean Absolute Error).

$$\frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

In discriminator, we use the valid matrix to determine whether the picture generated by the generator is fake or not. The valid matrix contains a series of continuous numbers, and we want to minimize the mean-square error between the valid matrix and true value: a 16 * 16 matrix which all elements in it are 0. We want the discriminator to identify all fake images.

In the generator, we use MAE as the loss function. MAE is quite similar to MSE. The formula is:

$$\mathrm{MAE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} |y_i - \hat{y}_i| .$$

We use this function to measure the difference between the fake image we generate and the original image.

From the formulas above, we can find that for MAE, if the output will be penalized more if it's far away from the true value. This is because we want the discriminator can 100% correctly recognize the fake image, so we extend the penalization for the unreasonable values. But for the generator, we don't need the image it generated exactly correct to the original image, it can have some small differences, no need to be so accurate, so we choose MAE rather than MSE.

Previous approaches have found it beneficial to mix the GAN objective with a more traditional loss, such as L2 distance. The discriminator's job remains unchanged, but the generator is tasked to not only fool the discriminator but also to be near the ground truth output in an L2 sense. We also explore this option, using L1 distance rather than L2 as L1 encourages less blurring:

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

Perceptual Error

If only use the CNN to train the model and use the loss function to measure the numerical error, you might encounter some situation that the numerical error is small but it is obviously unreasonable to the human's eye. To considering the perceptual error is also one of the reasons we choose the GAN to solve this problem. The mechanism of GAN(Generator and the discriminator fight against each other) can bring the measurement of the perceptual error to some extent. For more information about the mechanism of the GAN, you can go to the first part of the report.

Training the Model

Representing the problem is one thing, but can you train your model in a computationally tractable manner? What algorithms did you draw on? How did you determine convergence? How did you avoid overfitting?

Training Process

Generally speaking, we use the adversarial training method between G and D to achieve the colorization. We divide the training process into three parts, training discriminator D, training generator G, adversarial training. Corresponding to the functions mentioned in Methods for cGAN, I will introduce how it tractably computes in the training process together with the codes.

Load Training Data

```
for epoch in range(epochs):
    for batch_i, (imgs_A, imgs_B) in enumerate(self.data_loader.
load_train_batch(batch_size)):
```

For the project, we set batch size to be one which takes a single 256*256 image as train dataset in each batch. And it is because such batch size has been proved to be effective at image generation task [2]. In each batch, we load two images as train dataset, the real colorful image and its grayscale style image, described in Data part about their usage. And in each epoch, we train the model with all the 100 training images.

Training Discriminator

Target for D

Before explaining how to train the discriminator D, we want to describe the strategy of the training target for discriminator D. As we know, we want the discriminator to learn how to distinguish the ‘real’ colorful image and the ‘fake’ colorful image. For discriminator D, we choose to use SGD optimizer and MSE loss for training.

```
valid = np.ones((batch_size,) + self.disc_patch)
fake = np.zeros((batch_size,) + self.disc_patch)
```

So, we generated valid and fake arrays to represent if the image is fake or not at the size of the patch. For the valid array, we set all the elements to be one. In other words, when taking a ‘real’ colorful image as input, the target output array for D should be all ones. For the fake array, we set all the elements to be zero. In other words, when taking a ‘fake’ colorful image

as input, the target output array for D should be all zeros. And the patch size has been discussed in the Markovian discriminator (PatchGAN) Part.

D-Loss

```
d_loss_real = self.discriminator.train_on_batch([imgs_A, imgs_B], valid)
```

Then, we define two loss parameters d_loss_real, d_loss_fake to express the ability that the discriminator D distinguish the image is fake or real. To minimize the d_loss_real parameter, we use the real colorful image (imgs_A) and its grayscale image(imgs_B) as input and valid array with all 1s as the output target to train the discriminator D. It is corresponding to maximize logD(x,y) in formula (1). In other words, we use fake images to train the ability that the discriminator D distinguish the image is real.

```
d_loss_fake = self.discriminator.train_on_batch([fake_A, imgs_B], fake)
```

And to minimize the log(1-D(x,G(x,z))), we try to maximize the D(x,G(x,z)). In the training process, to minimize the d_loss_fake parameter, we use the colorized image(fake_A) together with its grayscale image(imgs_B) as input and the fake array with all 0s as output to train discriminator D. In other words, use fake images to train the ability that the discriminator D distinguish the image is fake.

```
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
```

Then we add up these two loss parameters as a total loss for the discriminator D, and our goal is to minimize the loss in training process to enhance D's ability to distinguish the fake or real images.

Training Generator

Combined Model

```
img_A = Input(shape=self.img_shape)
img_B = Input(shape=self.img_shape)
fake_A = self.generator(img_B)
self.discriminator.trainable = False
valid = self.discriminator([fake_A, img_B])
self.combined = Model(inputs=[img_A, img_B], outputs=[valid, fake_A])
```

Before describing how to train the generator, we want to introduce the combined model for training generator G. The combined model is used to train generator G. First, it takes the real colorful image (imgs_A) and its grayscale image(imgs_B) as input according to the cGAN. Then, it uses generator G to generate a colorized image(fake_A) based on the grayscale image(imgs_B). Next, we use the valid array to describe the validity of the generated image

with the output of discriminator D. Then we use the validity of the generated image(valid) together with the colorized image (fake_A) as output.

Target for G

Therefore, in the process of training generator G, when taking the grayscale image as conditional input, the output target of G should be a ‘real’ image with a valid array of all ones. For Generator G, we choose to use SGD optimizer and MAE, MSE losses for training.

G-Loss

```
g_loss = self.combined.train_on_batch([imgs_A, imgs_B], [valid,imgs_A])
```

And in the training process, we use valid arrays with all 1s together with the real colorful image (imgs_A) as the target to minimize the g_loss. In other words, we train the generator to generalize an image much similar to the real colorful image (imgs_A) to fool the discriminator D.

Adversarial Training

Mechanism

As we know, in cGAN we use the adversarial training method between G and D to achieve the colorization. More specifically, in the training process, the generator G learns to colorize a grayscale image to a color image which fools the discriminator D to be recognized as the real image. Meanwhile, we also train the discriminator D how to discriminate the ‘fake’ colorful image generated by G or the ‘real’ colorful image exists in our in reality.

And the training process of D and G is interweaving. Specifically, first we train D, and then alternately train G. For discriminator D, we choose to use SGD optimizer and MSE loss for training. For Generator G, we choose to use SGD optimizer and MAE, MSE losses for training. Despite the training process of D and G has been separated into each other, the training processes to minimize loss of G and D are mutual restraint to each other according to the internal mechanism of cGAN. Because G learns to generate an image looks more real, and D learns to distinguish the ‘fake’ image generated by G.

Avoid Overfitting

For discriminator D and generator G, we both use dropout layer with 0.5 as dropout rate to avoid the overfitting problem.

Assessing the Final Project

How good is your final program, and how can you determine that? How did you validate it? What is your program good at, and what could use improvement? Do your program’s mistakes ‘make sense’? What happens if you try to color images unlike

anything the program was trained on? What kind of models and approaches, potential improvements and fixes, might you consider if you had more time and resources?

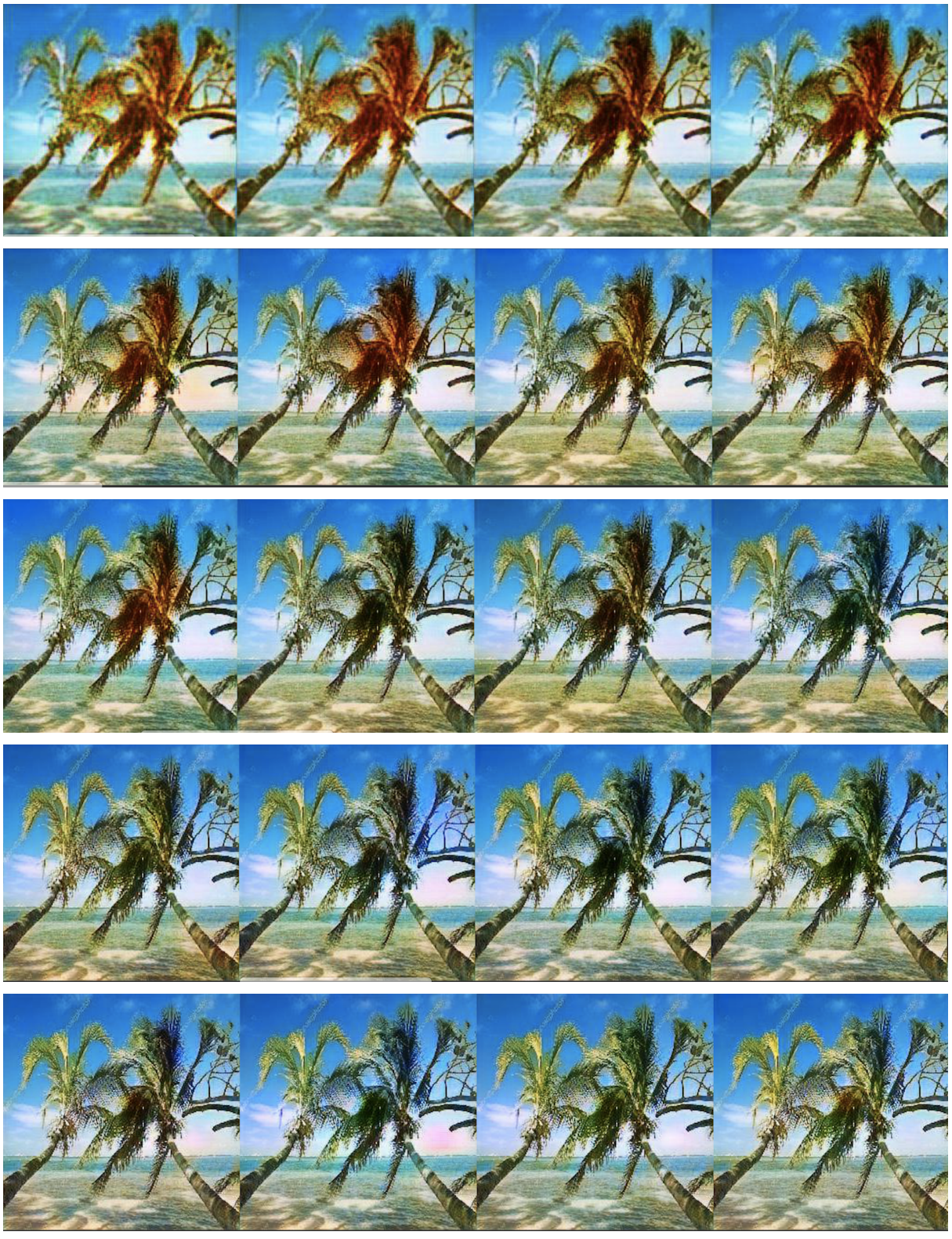
For our project, I ask 5 guys in my neighbor lab Multimedia Image Processing Lab by Prof. Marsic in Core 731 to evaluate the model. First I tell them to randomly choose 24 grayscale images from our test dataset and test with the trained model. Then I tell them to evaluate the colorized the image against the real colorful image. One each trail, each image appeared 5 seconds, after the image disappeared, they were given unlimited time to respond which image was fake. The test result together with their corresponding 'real' colorful images have been saved in./test_result/Final_test_result, you can check them by yourselves. Below is one of the colorizing result against the real colorful image.

Final Result:(fake is the image we geenrated)

Fake

Origin

We want to show the progress of our training through the gradual test results.



	1	2	3	4	5
1	1	0	0	0	1
2	0	0	0	0	0
3	0	0	1	0	1
4	1	1	0	0	1
5	0	0	0	0	0
6	1	1	1	0	1
7	0	1	0	0	0
8	0	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	1
11	0	0	0	0	0
12	1	1	0	0	1
13	0	0	1	0	1
14	0	1	0	0	0
15	0	1	0	0	0
16	0	0	0	0	0
17	1	1	1	1	0
18	0	0	0	0	0
19	0	0	1	1	1
20	1	0	0	0	0
21	0	0	0	0	1
22	0	0	0	1	0
23	1	0	1	0	0
24	0	0	0	1	1

Table 1: Identification results between real and fake images

In the excel form above, 1-5 stands for 5 volunteers to test our model. And 1-24 stands for 24 sets of test images colorized by our model. As mentioned before, ‘1’ in the form represents that volunteers choose to believe the image colorized by our model to be the ‘real’ image. ‘0’ means that volunteers choose to believe the real colorful image to be the ‘real’ image.

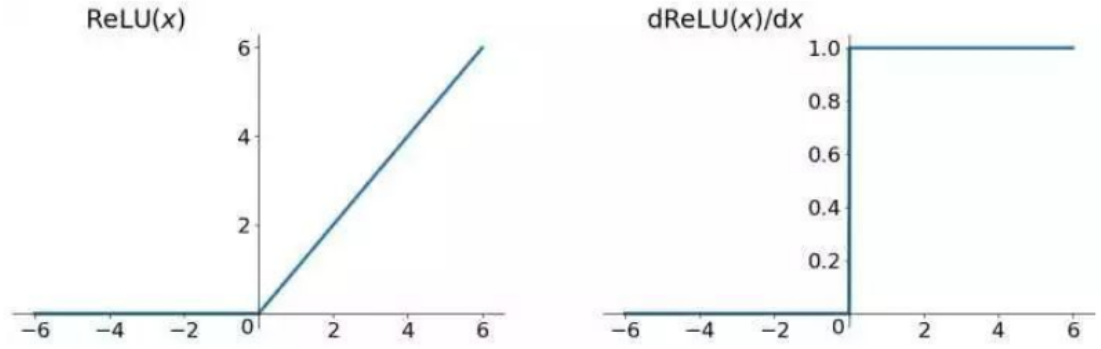
According to statistics, the above results turn out to be that given the colorized image together with the real image, people have a probability of 31.2% to believe that the colorized image should be the real image.

Our program is good at colorizing the image with beach, trees, and seas. For we only download 120 images with beach, trees, and seas from google image, so the colorizing result is only good at such images, but not good on other kinds of images. That makes sense. When coloring images, we load the trained model and only use the generator in cGAN generate the colorful images. Unlike the training process, we do not use the discriminator to determine if the image is fake or not. If we have more time, we may try more different loss functions and optimizer to train the model. And we also want to train the model with more kinds of images to make it more all-round to handle more kinds of images.

Supplementary Material

Activation Function

If you don't use an activation function (actually equivalent to an activation function is f(x) = x), in this case, your output each layer is a linear function of the last input layer, which is easy to verify. No matter how many layers your neural network have, the output is the linear combination of your inputs, which is equivalent to the effect of no hidden layer. This is the most primitive perception.



ReLU activation function is actually a function to get maximum value. Note that it is not full interval derivable, but we can take sub-gradient shown above. The function formula is:

$f(x) = max(0, x)$,

which is zero when x < 0 and then linear with slope 1 when x > 0. In other words, the activation is thresholded at zero.

Pros:

- Solved the problem of gradient vanishing (in the positive interval).
- The calculation speed is very fast, need to judge whether the input is greater than 0 or not.
- The convergence rate is much faster than sigmoid and tanh.

Cons:

- Dead ReLU Problem. It refers to the fact that certain neurons may never be activated, resulting in the corresponding parameters never being updated. During the training, ReLU units can "die". For example, a large gradient flowing through a ReLU neuron could cause the weights to update in such a way that the neuron will never activate on any data point again. If this happens, then the gradient flowing through the unit will forever be zero from that point on. If the learning rate is set too high, you may find that as much as 40% of your network can be "dead".

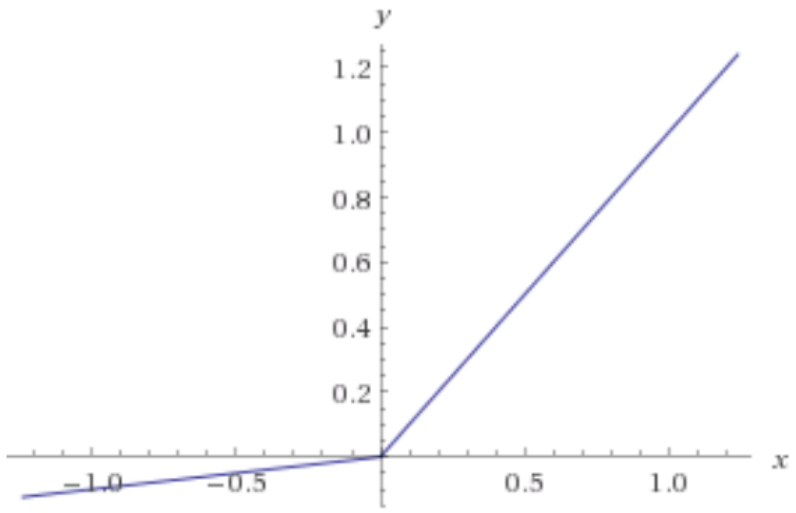
Why choose ReLU

First, when using sigmoid and other functions to calculate the activation function (exponential operation), the calculation is large. When calculating the error gradient by backpropagation, the derivation involves division, and the amount of calculation is relatively large. But using ReLU, the amount of calculation is saved a lot.

Second, for deep networks, when sigmoid functions propagate backward, it is easy for gradients to disappear (when sigmoid is near saturation, the transformation is too slow, and the derivative tends to zero, which will result in information lost, thus the training of deep networks cannot be completed)

Third, ReLU causes the output of a subset of neurons to be zero, which results in the sparseness of the interdependence of parameters, alleviating the occurrence of overfitting problems.

LeakyReLU



Leaky ReLUs is one attempt to fix the “dying ReLU” problem. Instead of being zero when x<0, a leaky ReLU will instead have a small negative slope of 0.01. The function formula is: f(x)=1(x<0)(αx)+1(x>=0)(x), where α is a small constant.
But the results are not always consistent. The slope in the negative region can also be made into a parameter of each neuron.

Batch Normalization

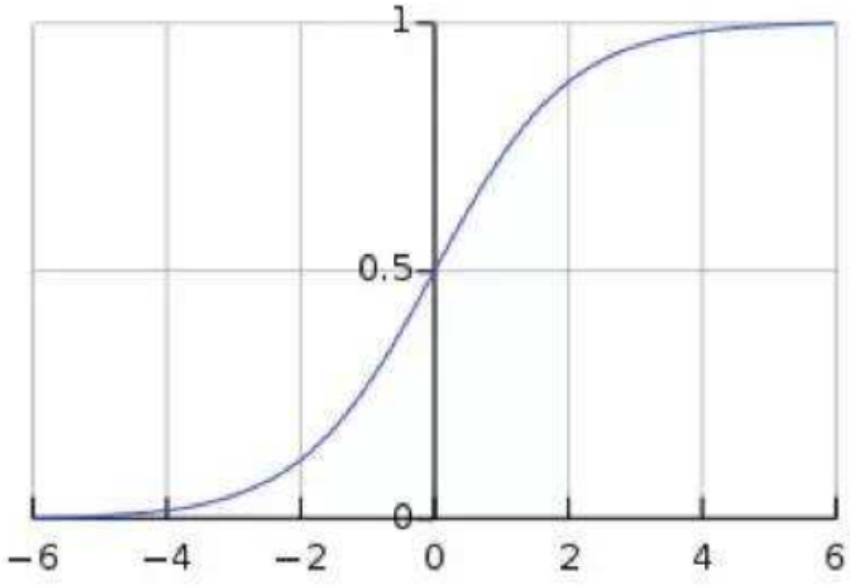
Batch Normalization is to solve “Internal Covariate Shift” problem[3]. Why the deeper the network is, the more difficult it is to train and the slower it converges? BN (Batch Normalization) essentially explains and solves this problem from a different angle.

For the network structure of deep learning, which contains many hidden layers, every hidden layer will face the problem of covariate shift because the parameters of each layer are constantly changing during the training process. That is to say, in the training process, the input distribution of the hidden layer always changes, which is called “Internal Covariate Shift”. Internal refers to the hidden layer of a deep network, which happens inside the network, rather than the covariate shift problem occurring only in the input layer.

So people introduce BN to make the activation input distribution of each hidden layer node be fixed.

Because the activation input value (x=WU+B, U is input) of the deep neural network before the nonlinear transformation is deepened as the network deepens, or during the training process, its distribution gradually shifts or changes. The reason why the training convergence is slow is that generally, the overall distribution gradually goes to the upper and lower limit of the nonlinear function (For Sigmoid function, it means that the activation input value WU+B is a large negative or positive value). Therefore, this leads to the disappearance of the gradient of the low-level neural network in the backpropagation, which is the essential reason for the slower convergence while training a deep neural network. And BN, through a certain standardization method, forcibly pulls the distribution of the input value of any neuron in each layer of the neural network back to the standard normal distribution with a mean of 0 and variance of 1. This causes the activation input value to fall in the region where the nonlinear function is sensitive to the input so that small changes in the input result in a large change in the loss function. This can avoid the gradient disappearance problem. It means that learning convergence is fast and can greatly speed up the training.

This is the figure of sigmoid(x):



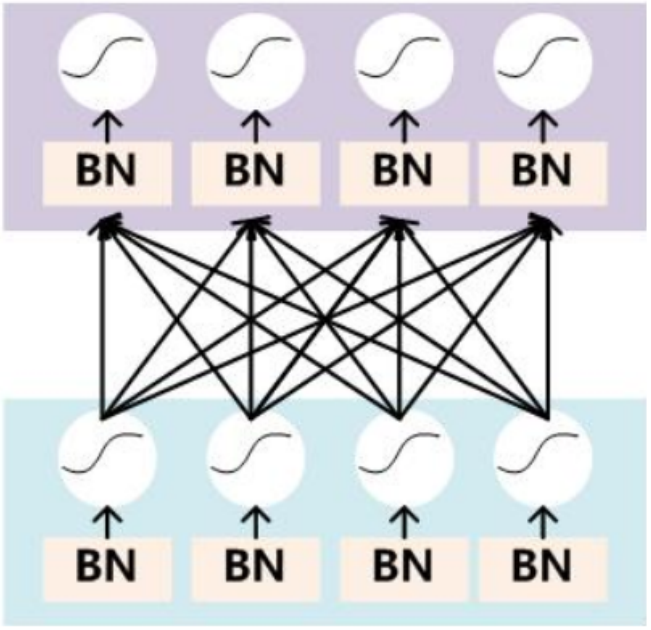
Assuming that the mean value of the original normal distribution of x before BN adjustment is -6, and the variance is 1, it means 95% of the values fall between [-8, -4], then the

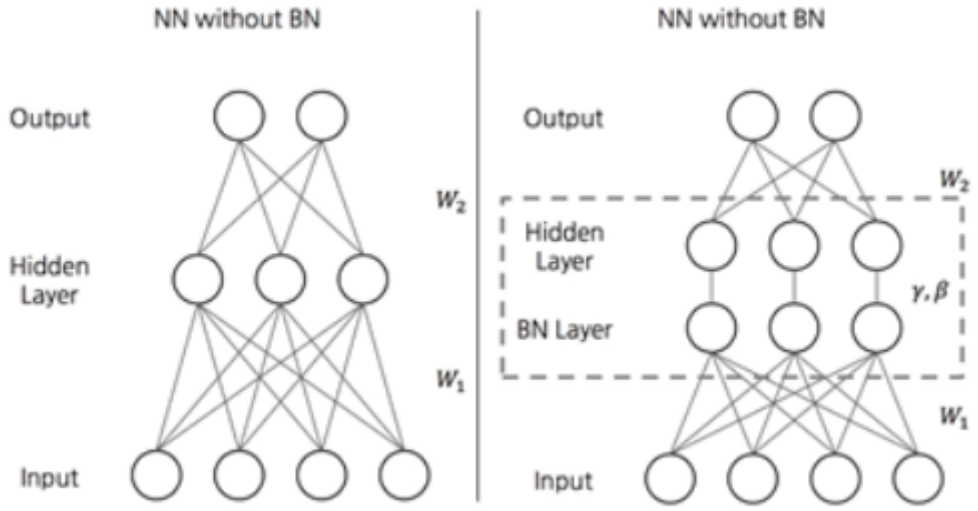
corresponding value of Sigmoid(x) function is close to 0. This is a typical gradient saturation region where the gradient changes very slowly.

And assuming that after BN, the mean value is 0, and the variance is 1, it means that 95% of the x value falls within the [-2, 2] range. This section is the region where the sigmoid(x) function is close to a linear transformation, which means that the gradient change is large.

As a result, after BN, most activation values fall into the linear region of the nonlinear function, and the corresponding derivatives are far from the saturated region of the derivatives, thus accelerating the training convergence process.

How to Batch Normalization?





For Mini-Batch SGD, there are m training instances in one training process. The specific BN operation is to transform the activation value of each neuron in the hidden layer as follows:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}}$$

To prevent the expression ability of network decline, each neuron adds two adjusting parameters (scale and shift), which are learned through training. These two parameters are used to inverse transform the transformed activation, which enhances the network expression ability.

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Inference

Replace the mean and variance statistics obtained from m training instances in Mini-Batch with statistics obtained from all training instances. When reasoning, we can use global statistics directly.

So how to get mean value and variance. Every time we do Mini-Batch training, there will be the mean and variance of the Mini-Batch training instances. Now we need global statistics, we only need to remember the mean and variance statistics of each Mini-Batch, then, we can get the global statistics by calculating the corresponding mathematical expectations of these mean and variance. As follows:

$$E[x] \leftarrow E_{\mathrm{B}}[\mu_{\mathrm{B}}]$$

$$Var[x] \leftarrow \frac{m}{m-1} E_{\mathrm{B}}[\sigma_{\mathrm{B}}^2]$$

With mean and variance, each hidden neuron has corresponding trained scaling parameters and shift parameters, so that the activation data of each neuron can be transformed to calculate BN in the process of inference. In the following way:

$$y = \frac{\gamma}{\sqrt{Var[x] + \varepsilon}} \cdot x + (\beta - \frac{\gamma \cdot E[x]}{\sqrt{Var[x] + \varepsilon}})$$

This formula is actually equivalent to the training formula:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

For each hidden neuron:

$$\frac{\gamma}{\sqrt{Var[x] + \varepsilon}} \qquad \frac{\gamma \cdot E[x]}{\sqrt{Var[x] + \varepsilon}}$$

They are all fixed values so that these two values can be stored in advance, and they can be used directly in the reasoning. If the number of hidden layer neurons is large, the amount of computation saved is much more.

Pros.

- The training speed is greatly improved, and the convergence process is greatly accelerated.
- Increase the classification effect.
- The process of parameter adjustment is much simpler, the requirement is much simpler, the requirement of initialization is not so high, and a large learning rate can be used.

Reference:

[1] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Gen- erative adversarial nets. In NIPS, 2014.
[2]. D. Ulyanov, A. Vedaldi, and V. Lempitsky. Instance normal- ization: The missing ingredient for fast stylization. arXiv preprint arXiv:1607.08022, 2016.
[3] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv preprint arXiv:1502.03167.