Group Member:
Xin Yang(xy213)
Zhuohang Li(zl299)

CS 520: Assignment 4 - Colorization

1. Artificial Neural Network Based Approach

1.1. Introduction

We first tried an ANN-based approach to colorize grayscale pictures. We choose Python as the programming language and this part is done in the environment of python 3.6. The ANN is implemented by ourselves only using one external package - Numpy. To read in pixel data from files and show the generated pictures, we use two 3rd-party libraries: skimage and matplotlib.

1.2. Representation

For the ANN-based approach, our idea was to simplify the problem from a regression problem to a discrete classification problem. Instead of using true color, we pick a fixed number of colors to illustrate. We use a window of grayscale values to determine the RGB value of a single pixel. Sliding the window along the grayscale image would give us a generated colorized image.

1.3. Data

Images are stored as a two-dimensional array of pixels. The length and width of the array correspond to the number of pixels in the image. A pixel is represented by a one-dimensional array in the format of [R, G, B]. The RGB value of a true color image range from 0 to 255. To simplify the problem, we set RGB values to be four discrete values: 0, 64, 128, 192, which can be labeled as 0, 1, 2, 3 respectively. We pre-process the original image pixel by pixel to round its RGB to the nearest value. To determine the color of a pixel, we use a 3*3 window of grayscale value to map to an RGB vector, which represents

the prediction for the central pixel. To further simplify the problem, we choose to build three separate neural networks to calculate the RGB value respectively. Each neural network is designed to have 3 layers, the first and second layer is consist of 9 neurons and the third layer has 4. The reason being that the first layer needs to match the format of input data, which is a 1*9 vector. The third layer is the output layer, it is the part to vote towards a most favorable value. The likely values are 0, 64, 128 and 192, which can be mapping to 0, 1, 2, 3. To increase robustness, we format 0-3 to be one-hot code:

Decimal	One-hot code
0	[1, 0, 0, 0]
1	[0, 1, 0, 0]
2	[0, 0, 1, 0]
3	[0, 0, 0, 1]

So we need 4 neurons in the last layer to generate the output one-hot code.

1.4. Evaluation

To evaluate our result quantitatively, we introduce the difference between the input image and the generated image. The difference of two pixels is given by:

$$d = \frac{|(R_1 - R_2) + (G_1 - G_2) + (B_1 - B_2)|}{S}$$

Where S is the stride between discrete RGB values, in our case, S = 64.

The difference of two given images is the average difference for all pixels. The value represents shows the magnitude of the distortion of the generated image, comparing to the original one. A larger number indicates bigger distortion which means a bad prediction. Our experiment shows that a difference at around 0.80 should normally deliver a good simulation.

1.5. Training

The structure of our neural network is 9 neurons in the input layer, 9 neurons in the hidden layer, and 4 neurons in the output layer. Without loss of generality, we choose to use the left half of an image as the training set, and the right half as the test set. For images with poor bilateral symmetry, our program can be easily modified to use another similar image as training set instead. We first convert the left half of image to grayscale, which is our training input. Then we rescale the value of R channel of the left half image, transform it to one-hot code to serve as the training label. Then we start training our network. We use a window with size 3*3 and slide it along the grayscale image. The 9 grayscale values within the window, labeled with the R channel value of the central pixel, are flattened into a 1*9 vector and fed into the network. We use hyperparameters such as the number of epochs and learning rate to control the training process and to find a convergence. We use the same process to train another two networks for G and B channel. After the training, our network is ready to make a prediction. We feed the grayscale image from the test set into the network. The output layer will vote and generate a one-hot code mapping to 0-3. The output decimal is then multiplied by a factor of 64 to produce an RGB value. Combining all the pixels and three channels we could generate an image, which is the prediction result.

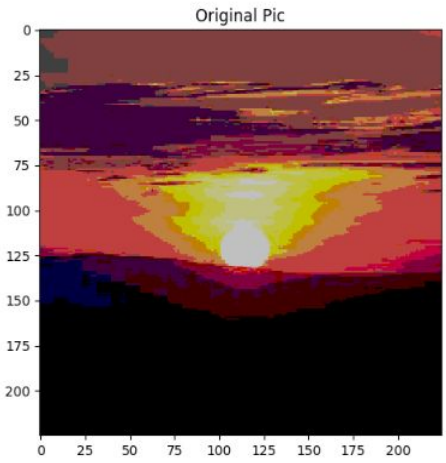
1.6. Assessment

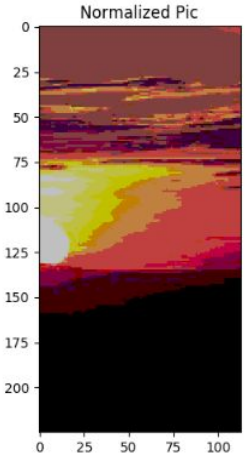
We can assess our program both subjectively and quantitatively. Below are some of our results. For each set of images, the upper left is the original picture after pre-processing. The lower right is the generated image of our program. As a comparison, the lower left is the ideal output, which is the right half of the original picture. The differences are coffee: 0.67, sunset: 0.83 and beach: 1.17, which is consistent with our subjective feelings.

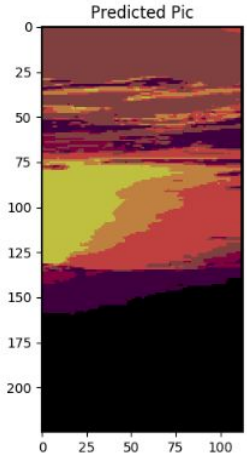
Original Pic

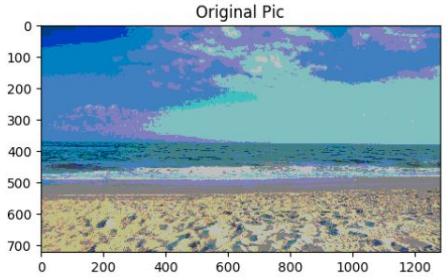
Normalized Pic

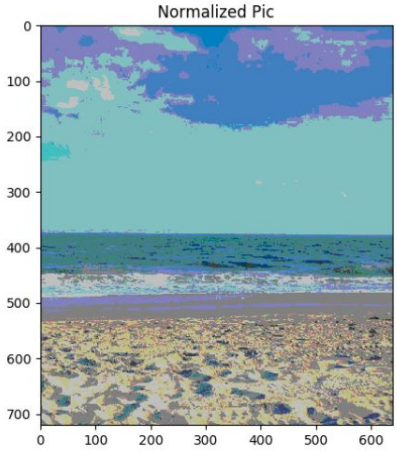
Predicted Pic

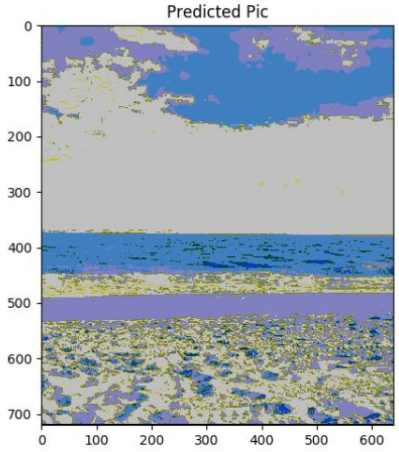

Original Pic
0
25
50
75
100
125
150
175
200
0
25
50
75
100
125
150
175
200


Normalized Pic
0
25
50
75
100
125
150
175
200
0
25
50
75
100

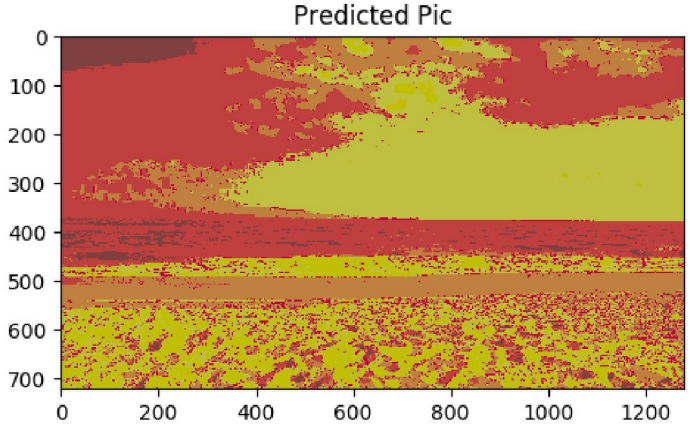

Predicted Pic
0
25
50
75
100
125
150
175
200
0
25
50
75
100







We also run an experiment to use the sunset picture to train the network, and try to predict the beach image, below is what we got:



We can see that the result is terrible but in a sensible way because it actually shows the exact color pattern of the training data.
Due to the fact that our algorithm is taking a 3*3 window of pixels each time, the program can be easily applied to any image without being limited by the size and quality of the image. On the other hand,

if the image is consist of different color areas with a similar grayscale pattern, the program is likely to be confused and perform the wrong prediction e.g. in the beach picture above, the sky and the beach was painted with a similar color in the generated image. This could be improved by enlarging the size of the window to pick more surrounding information before making the decision on a pixel. Another improvement could be to use Momentum / RMSprop / Adam algorithm instead of gradient descent, to accelerate the convergence. We could also use a wider range of RGB values to produce more vivid images.

2. A Convolutional Neural Network Approach

2.1. Introduction

To better extract the feature hidden in the picture, we also tried a convolutional neural network based on the TensorFlow framework. The image processing library we use includes the Pillow and Matplotlib. We also followed some instructions and implemented the methods to convert between the LAB color space and RGB color space.

The basic workflow of our CNN approach is: First we map the original picture to a LAB color space, then use the L channel as the grayscale input, and output the predicted value of processed channel A and B data. Then we recover the prediction to the original LAB channel values, and revert to RGB and plot them for evaluation.

2.2. Representation

Our program is based on Python because it needs lots of matrix computation. Our data is thus represented by Numpy arrays. The color space we select here is the CIELAB space(LAB). Different from the RGB space, the LAB space consists of three channels, and each channel, i.e., L, A, B represent the lightness, ranging from 0 to 100, green-red, and blue-yellow, both ranging from -128 to 127. The

advantage is that the lightness channel is right the grayscale image, although would be slightly different due to the index range. We can directly take the channel L as our input feature, and the dimension of prediction can be decreased to 2-D (Channel A and Channel B). The prediction result adding the input feature is a complete LAB mode image, which can then be converted to the standard RGB space. Note that there is no direct conversion between LAB and RGB, an intermediate XYZ space is required. The LAB mode is more convenient for the computer to adjust the image, and it has a larger color space comparing with the RGB, i.e., we can achieve a more flexible adjustment with even fewer parameters.

The convolution process can extract the feature in a more 'image' way and can get more information in a higher dimension comparing with the traditional neural network.

2.3. Data

The dataset we use is from the Google Image Search. We choose three scenarios as the evaluation, which are the beach, sunrise, and forest. Our dataset is included in the folder, each dataset contains around 15 images, and are separated into two parts: the training set and testing set by a ratio of 7:3. All the scenarios we choose contains a limited amount of colors, e.g., the beach will only contain the blue for the ocean, yellow for the beach, and a little green for the trees. In this manner, it can reduce the difficulty properly for the program and would thus require less computational resources. The images in a single category are considered very similar in a human way, either in terms of the color and the complexity of objects. The dataset we choose can guarantee the convergence in some perspective.

The algorithm is a regression rather than classification. We suggest the regression can show a more vivid color comparing with the classification using our limited resources because a classification will

need to significantly cut down the number of colors that we use, and this is a less general way, although the classification can help save a lot of computation capability, we still prefer a more colorful way.

We first resize our dataset since the CNN asks for a fixed size of the input. We read the image files from the disk and convert to RGB space tensors. Then we convert to the LAB space according to the existing algorithms. Our hardware platform is on a GPU accelerated PC(NVIDIA Geforce GTX 1070) running x64 Linux OS, which has 8 GB Graphics Memory. After several trials and errors, we decide to resize our images to $64 * 64$ (pixels) to balance the readability and computation power.

Since the convolutional neural network extracts the feature on a two dimensional way, we represent the input data as a 2d array, the elements of the array represent the grayscale value in the coordinate. We set the labels and output to be one-dimensional arrays. The content of the dimensional array is the flattened values of channel A and channel B, i.e., the first half is all the values in channel A, the rest stand for the ones in channel B. In this way, we can easily reshape the prediction result to a LAB channel, meanwhile we can still perform the regularization and reduce method that we used to calculate the loss.

2.4. Evaluation

According to the way we represent our labels, the relevant element in the prediction array and label array represents the color of one pixel. We can represent the loss using the Euclidean distance.

The distance between a prediction tuple(pred_A, pred_B) and the actual label (A, B):

$$\sqrt{(A - pred\_A)^2 + (B - pred\_B)^2}$$

In order to make the punishment larger, we removed the square root of the function, and the distance is simplified to:

$$(A - pred\_A)^2 + (B - pred\_B)^2$$

Also considering the fact that the channel A and channel B are all in the same flattened array. The loss can be represented as below, i stands for the index of channel A and j stands for the index of channel B, the relation of i and j can be represented as below, the width and height stand for how many pixels we have, and in this case is $64 * 64$:

$$(Label_i - Pred_i)^2 + (Label_j - Pred_j)^2, \quad (j - i = width * height)$$

The goal is to minimize the distance between our prediction and label, so we calculated the mean value of all the pixels:

$$\frac{\sum_{i=0}^{n} (Label_i - Pred_i)^2}{n}$$

The n is the total number of pixels in the image, and our loss function is basically a mean square error (MSE).

The accuracy is a challenge for us because we have very limited experience to the LAB color space. After observing the results of prediction output, we set the judgment standard as if the final prediction value and the real value has an error less than ten, we consider it as correct, otherwise, we think this is too far from our visual sense. Also when we perform our experiments, we are very careful that this accuracy only has very limited reference value, usually an over 60% accuracy is ideal enough for a human to tell the similarity.

2.5. Training

Note that the original value of channel A and B are from -128 to 127, but the activation function we choose is ReLU, which is not centered

at 0. In this case, we simply added 128 and make it ranging from 0 to 256.
At the output layer, to make our prediction more effective and easy to converge, we first regularize the result to the range [0,1], and we multiply the regularized value with 255 to keep a consistent range as our labels.

The structure of our CNN is described as follows:
Input Layer -> Convolutional Layer -> Pooling Layer -> Convolutional Layer -> Pooling Layer -> Fully Connected Layer -> Fully Connected Layer -> Dropout Layer -> Output Layer

The convolution kernel we use has a 5*5 size with the same padding mode, which will fill the edge of the image if the surrounding pixels do not exist. All the pooling layers did a max pooling by size 2*2, which means the pooling process will take each 2*2 area as input, and output the max value of the four numbers. The next three fully connected layers flattened the output of the second pooling layer, and take a dropout layer to decrease the overfitting. The activation function we choose is the Rectified Linear Unit(ReLU), which would set the values below 0 as 0, and keep the values greater than 0 as itself. ReLU is widely adopted in the Convolutional Neural Network in that it can non-linearize the output of the neural network. Besides, comparing with the traditional tanh and sigmoid, it can also help simplify the computation in the backpropagation and help prevent the gradient vanishing(information could get lost when it comes to the saturation). The output of the first convolutional layer is 32, and for the second convolutional layer is 64. The flattened output of the second convolutional layer will have width*height*4 neurons. The number of neurons of the first fully connected layer is 2048, and for the second fully connected layer is 4096. As for the output layer, the shape is width*height*2. We can see that to fit the size of the total

pixels, we significantly increased the number of neurons compared with relatively more basic image tasks like the most famous MNIST.

Our core algorithm is the gradient descent algorithm. In our example, we set a slow learning rate 0.0001, and a big training epoch number, which is around 8000. Also note that since the size of our picture is big(64*64), we only randomly select a small batch of images(3 images as a batch) from our training set, otherwise we could easily get an Out-of-Memory Error(OOM). In the meanwhile, this idea is also called Mini-Batch Gradient Descent(MBGD). Comparing with the traditional Gradient Descent algorithm, which put all the training set into each training iteration, and the Batch Gradient Descent, which only put one training sample into each iteration, the MBGD combines the advantages of both algorithms, and can better find the global minimum with a fast convergence speed. The dropout layer we mostly will not use it as we have a great number of neurons, the dropout can make the convergence harder.

We didn't set a standard to stop the training process, after our experiments and observation, the iterations that the program takes might vary and according to our accuracy evaluation strategy, the overall result also might vary based on different datasets. Usually, we would determine the program reached a convergence when the training accuracy and test accuracy all meet a relatively high score, e.g., around 60%, and after this period, the accuracy of training set will get significantly higher but for test cases will heavily decrease, which is the phenomenon of overfitting. Usually, the convergence will take around 5000-8000 epochs based on our configurations.

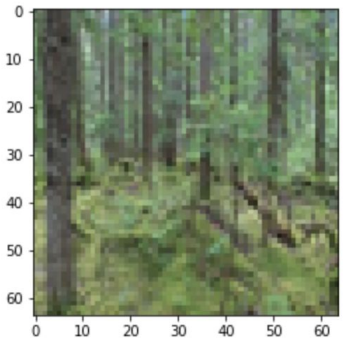
2.6. Assessment

Based on our evaluation standard, we consider a pixel to be accurate if the prediction value has an error less than 10 with the actual value in a

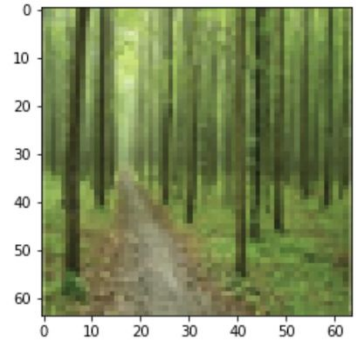
LAB space. Our program could achieve an accuracy of 57% for the forest dataset, and 64% for the beach, 54% for the sunrise. Examples of the prediction of the test cases are as the pictures below. Note that the test cases are never used in the training progress.

Forest:

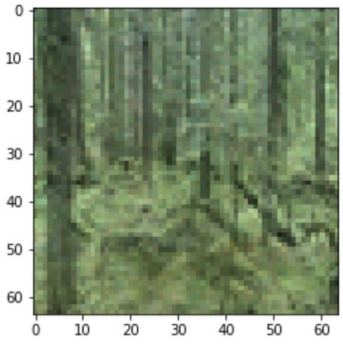
RGB:



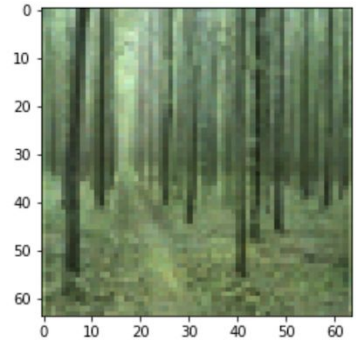
RGB:



Prediction:

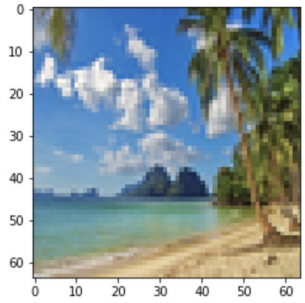


Prediction:

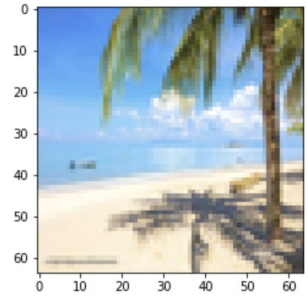


Beach:

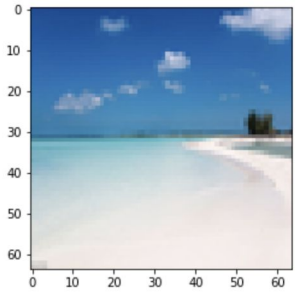
RGB:



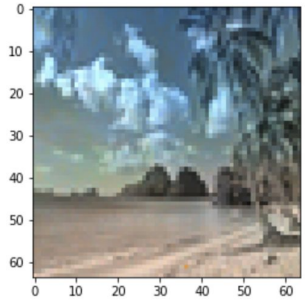
RGB:



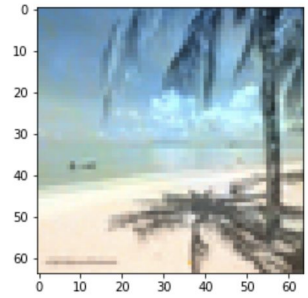
RGB:



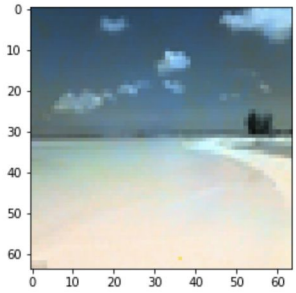
Prediction:



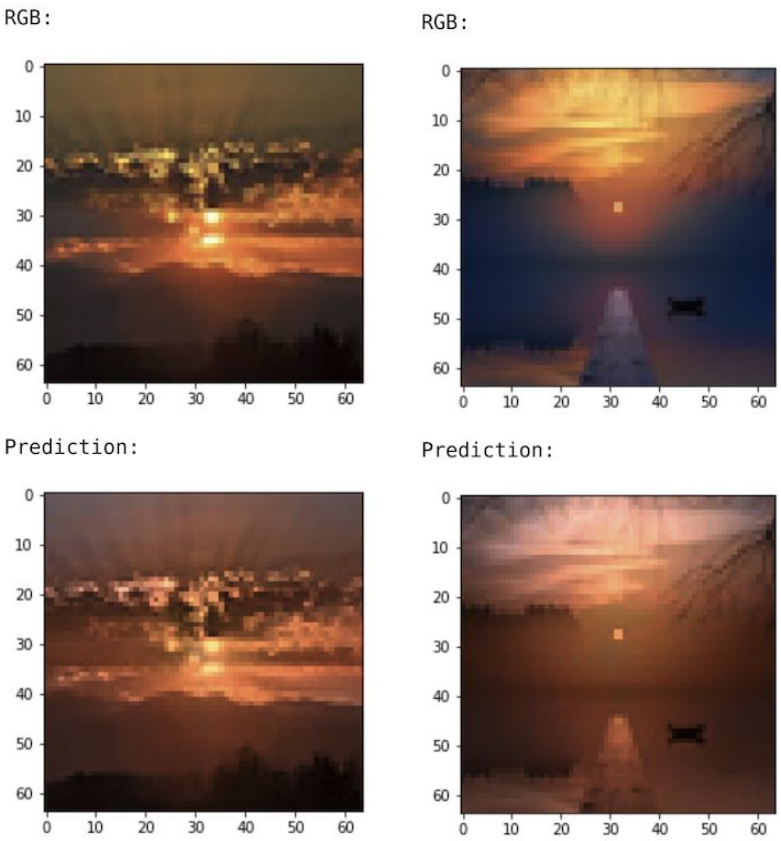
Prediction:



Prediction:



Sunrise:



From the results above, we can infer that the convolutional network can extract the main features of the scenarios correctly, but for objects that are not commonly existing, the CNN failed to colorize the correct color, e.g., the tree in the beach scenario and the wooden walkway in the sunrise scenario. As a human, I think our program can correctly recognize the picture and the colorized result is fair enough that we won't feel wired. Signs of overfitting still exist, from the results of the beach scenario, we can see that the program tends to colorize the above half to be blue, and the below half as yellow, which meets our common sense but doesn't fit some images, like the third one, it colorizes part of the curved beach to be blue and mistaken it as the sky or the sea. And for the forest scenario, the dark brown color of the mud road and tree trunks are hard to be seen. Most of the pixels are just colored to be a green since the green takes the majority. This mistake made by the program can be understood since our loss

function is to try to minimize the average error of each color channel of each pixel. By mistakenly colorizing some small objects in the picture can increase the overall score and achieve a more general colorize method, thus meet the convergence standard that we made: we stop when the prediction accuracy and training accuracy both get a high percentage, this means the strategy we want is a more general way, and this has to sacrifice the performance of small objects with a minor color that is different from the dominant hue. Overall, the prediction results meet the common sense of our knowledge and showed a relatively dull color range.

Comparing with the first neural network approach, the convolutional neural network shows a better performance and a more vivid color space. The reasons are various, not only we change the strategy from a classification to a regression. The convolutional process can also better extract the feature of a two-dimensional image array and has a more general application. But the costs are also obvious: far more computation resources are required to perform the convolutional process, and the number of training sets also increases in order to get a better result. For an application without accurate and wide color range demands, the traditional neural network can be a more efficient solution.