CS 520: Assignment 4 - Colorization

Haoyang Zhang, Han Wu, Shengjie Li, Zhichao Xu

December 15, 2018

1 Introduction, group members and division of workload

In this group project, we implemented a image colorizer which can generate satisfying color images given grayscale images.

Name RUID	Workload
Haoyang Zhang 188008687	Implemented a toy CNN based on numpy. Wrote Usage.html and HowToBuildACNN.html which are two documents about the toy CNN. Wrote part of the report. Analyzed the performance of this model on images belonging to some classes that have never been trained before.
Han Wu 189008460	Wrote part of the report and participated in the discussion. Analyzed the outcome of our colorizer.
Shengjie Li 188008047	Implemented the colorizer. Trained the model. Wrote part of the report. Finished the format design of the report using LaTeX.
Zhichao Xu 188008912	Wrote part of the report. Came up with the original structure of network.

2 The Problem

- **Representing the Process:**

  1. **How can you represent the coloring process in a way that a computer can handle?**
     A color image can be interpreted as an array with 3 channels – channel R, channel G, channel B. A grayscale image can be interpreted as a color image being compressed to having only one channel gray, by the formula $Gray(r, g, b) = 0.21r + 0.72g + 0.07b$. So the coloring process can be seen as: given a one-channel image, produce a three-channel image. To achieve this, we are using a convolutional autoencoder – An autoencoder with convolutional layers in its encoding part and deconvolutional layers in its decoding part.
  2. **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).

     Every image of the input is a $3 \times 32 \times 32$ numpy array – RGB values of every pixel. Each RGB value is between [0, 255]. We will first pre-process the data, get the grayscale of the images, standardize the value, make every image to be a $1 \times 32 \times 32$ numpy array with values between [0, 1].

     Because we are using tanh as our activation function. So the ouput of the network is a $3 \times 32 \times 32$ array with RGB channels and values between [-1, 1]. We scale it between [0, 1] and

use this value to calculate the loss. If we are going to plot the results, we can simply multiply every value by 255.

Because of the dense layers, each output $3 \times 1 \times 1$ is related to the whole $1 \times 32 \times 32$ input graph

- **Data:**

  1. **Where are you getting your data from to train/build your model?**

     We are using the The CIFAR-10 dataset.

     This dataset consists of 60,000 32×32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images in total. The dataset is divided into five training batches and one test batch, each with 10,000 images. The test batch contains exactly 1,000 randomly-selected images from each class.

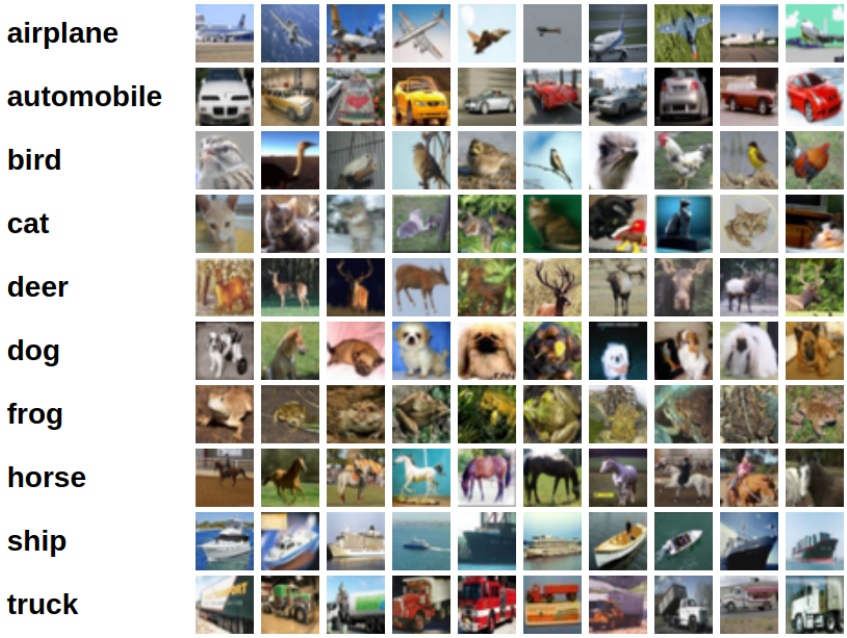

Figure 1: The CIFAR-10 dataset

  2. **What kind of pre-processing might you consider doing?**

     Each training batch is a 10000×3072 numpy array. Each row of the array stores a 32×32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The value of each channel is between $[0, 255]$. We are going to scale the RGB channel from $[0, 255]$ to $[0, 1]$. By standardizing the data, we could train faster and reduce the chance of getting stuck in a local optima.

- **Our Model:**

  Our model is an basically autoencoder with convolutional layers in the encoding part and deconvolutional layers in the decoding part.

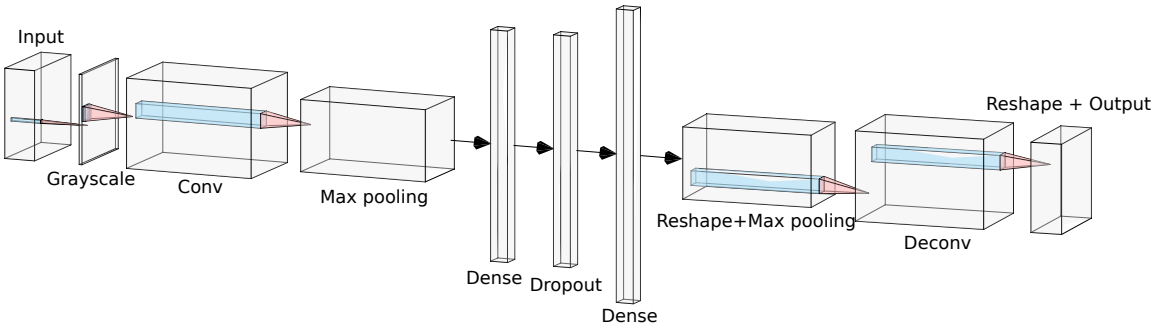

Figure 2: Structure of our model

1. **Input layer:** uses 4d-tensor to represent the input image.
2. **Grayscale layer:** calculates the grayscale of the input colored image.
3. **Convolutional layer:** in this layer we are using a filter of size 5×5. During the forward pass, we convolute each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. In this process, we will produce a 2-d activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual features, such as an edge of some orientation or a blotch of some color on the first layer, or eventually some wheel-like patterns on higher layers of the network. We will stack these activation maps along the depth dimension and produce the output volume.
4. **Max Pooling layer:** this function of this layer is, progressively reducing the spatial size of the representation to reduce the amount, if parameters and computation in the network, and hence to also control overfitting. This layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The window size of this pooling layer is 2×2.
5. **Dense layer:** in this layer the neurons have full connections to all activations in the previous layer. Their activations can hence be computed with a matrix multiplication followed by a bias offset.
6. **Dropout layer:** this layer is a regularizer that randomly sets input values to zero with a probability of 30%. It is used to prevent overfitting.
7. **Dense layer:** another fully connected layer.
8. **Reshape + Max Unpooling layer:** first the reshape layer reshape the data, then the max unpooling layer performs unpooling over the last two dimensions of a 4D tensor. This layer is essential if we want to reconstruct the image.
9. **Deconvolutional layer:** This layer is used for upsampling (backwards strided convolution). It changes the shape from $100 \times 32 \times 32$ to $3 \times 32 \times 32$ which is the shape we want for an image. It can coarse outputs to dense pixels by interpolation. In-network upsampling is fast and effective for learning dense prediction.
10. **Reshape + Output layer:** this layer reshapes the network and outputs the generated image.

- **Evaluating the Model:**

1. **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?**
   For the evaluation part, we used the squared loss as the criteria.

$$\mathcal{L} = ||output - input||^2$$

   When the loss is small enough, we can know our model is performing good.

2. **How can you assess the error of your model (hopefully in a way that can be learned from)?**
   - From the numerical perspective, the mean squared validation loss is, the better the model is performing.
   - From the perceptual perspective, the closer the color is to the original picture, the better the model is performing.

- **Training the Model:**

1. Representing the problem is one thing, but can you train your model in a computationally tractable manner?

   It is actually quite easy to train our model because of the small size of the input. Generally, for each iteration, we would need to train for 40 seconds using a GTX 1070ti, for 25 seconds using a GTX 1080ti. We could achieve a good result in less than 100 iterations of training most of the time.

2. What algorithms did you draw on?

   We are using Stochastic Gradient Descent algorithm with Nestrerov momentum, and we are using back propagation to update the gradient.

3. How did you determine convergence?

   We validate the model and print out the validation loss every 10 iterations. When the validation loss stabilizes and even starts to rise, we stop our training and call it a convergence.

4. How did you avoid overfitting?

   We are using Max Pooling layers and Dropout Layers in our model to prevent overfitting.
   - Max pooling is basically selecting a subset of features, so we are less likely to be always training on false patterns.
   - Dropout layers randomly drop some components of our neural network with some probability, which actually creates a different network. Thus after we finished the training, we actually have an ensemble of models. The probability is typically 50%, but after our experiments, we found 30% is better.

- **Assessing the Final Project:**

1. How good is your final program, and how can you determine that? How did you validate it?

Figure 3: Some results

Our final model is satisfying as Fig 3 shows.

We use our validation dataset to calculate the validation loss. The validation loss of our final model achieved 20.928495, which is reasonable to us.

2. What is your program good at, and what could use improvement?

   Our program is performing very good on colorizing animals and vehicles if they are not blue or red. When it comes to images with blue and red colors, our program could totally mess up.

3. Do your program's mistakes 'make sense'?

Figure 4: Some results

The colorizer can totally mess up blue and red or white and yellow as Fig 4 shows. But to be honest, we cannot tell the color of the car by just looking at its grayscale image. We think this kind of mistakes are quite resonable.

4. What happens if you try to color images unlike anything the program was trained on?

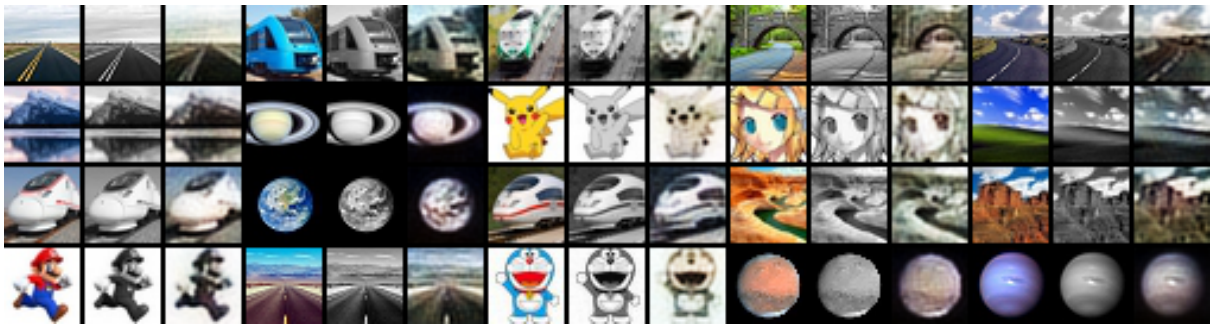

Figure 5: Some results

Our dataset contains images from 10 classes – airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. Thus, to verify the performance of this question, we chose a total of 20 images with 5 classes – Roads, Planets, Trains, Landscapes and Anime Characters.

The average loss of all 5 above classes is 60.927757, which is a lot higher compared with a loss of about 20 while validating on trained classes.

- For Roads, it performs not bad because a grayscale picture of roads still provides lots of information: roads are dark, lines are light, and sky is nearly white. And since we have trained many car and truck pictures, which contains some roads. It is not suppressing that it performs well.
- For Trains, some colorized pictures are not that bad compared with the original ones, except that sometimes it colors the train body with a different color. It is reasonable that a train can be either blue or dark yellow. Since there are some similarities between trucks and trains. We can conclude our model have sort of ability of generalizing.
- Planets are nightmare, but how could we know a planet is red, blue or any other color? They are all round and with nearly the same gray scale.
- Many mistakes are made when it tries to color images of landscape, but an interesting thing is that it knows all mountains should color brown. Though some mountains are not brown. Nevertheless, it seems have sort of idea of "Mountains".
- Our program failed on all images of Anime Characters. All training images we have are natural and do not contain any human-like creature. Therefore the failure is acceptable as we expect its failure. But if we have trained on images of human faces, our program might have performed better on Anime faces.

5. What kind of models and approaches, potential improvements and fixes, might you consider if you had more time and resources?

   For the potential improvement, we are considering using larger pictures and larger dataset, to increase our model's generalization ability. Also, we are considering using deeper convolutional layers, which has been proven of having strong ability in image processing.

3 Our understanding towards CNN

To demonstrate our understanding towards supervised learning, we implemented a CNN by ourself. In our CNN, we have implemented functions that correspond to every API of third-party libraries we used in the project(this correspondence can be find in 'toyCNN/docs/Usage.html'). We do not have any idea on how to properly insert this part of content into this report, therefore please refer to 'toyCNN/docs/Usage.html' and 'toyCNN/docs/HowToBuildACNN.html' for more information. Thank you!