Introduction

Steganography is a way for people to hide some important information into other pictures, documents, messages, and so on. Different from other methods like cryptography, steganography is more discreet but the information concealed by steganography is more likely to be revealed by other people. In the project, we solve steganography by using two different methods: the Least Significant Bit (LSB) and Neural Network (NN). By comparing the results obtained from LSB and NN, we conclude which one is better.

Step-by-step Description

Task 1 (LSB):

There are two parts: Hiding an image into another image and revealing the hidden image.

Hiding an image into another image:

1. We download a few images from the test data set using the link http://r0k.us/graphics/kodak/ , and then transfer them into RGB tuple.
2. By using the method called "__int_to_bin", we convert integers from 0-225 to the format of binary tuples.
3. Through employing the "__merge_rgb" method, we merge two images. The first thing we do is to check the dimension of two images. Because the dimension of the two images are the same, we extract the first 4 bits of the two images for each pixel. Next, we combine the first 4 bits of cover image with the first 4 bits of secret image. Then we get a new number with 8 bits.
4. We convert the new binary value we got from the last step into a decimal value (from 0 to 255) and then we get an merged image.

Revealing the hidden image:

1. We define a function named “unmerge(img)” to reveal the secret image. First, it is important for us to identify the number of bits for the hidden image.
2. We use two four loops to run through all of the pixels of the image and then get the RGB of the image.
3. We separate the first 4 bits of the image from the last 4 bits of the image.
4. By combining the first 4 bits of the image with “1000”, we can get the cover image.
5. By combining the last 4 bits of the image with “1000”, we can get the secret image.

Results:

unmerged_cover

emerged_secret

For task 1, we used the LSB to solve steganography and we did the task very well. During the process, we did not encounter any questions. Because we spent a large amount of time reading the paper and understanding the code listed under the description of the project. We found that the article and code are very important and helped us a lot in solving the task.

Task 2 (NN):

1. The first thing we do is to upload a file named “Kaggle.json” and then import the dataset into colab. (We put the json file in repository)
2. At first, we did not know what format of image we should transfer. After reading some example codes and observing the features of the dataset imported into colab, we decided to create the path name of files using ‘for loop’. Because the parameter of modules is the path name of files in most situations, it is convenient for us to use some modules directly.
3. For the “Load data function,” we used “os.listdir” to check step-by-step and find all files under a folder. Then we combine to create a valid path name by employing

"os.join.path." Because there is a file with 'db' format in each folder, we use "if condition" to extract the file with 'db' format in order to make sure that all the elements in the list are images(or images type). We did a lot of image processing in this step, including normalizing the image by dividing 256 and transforming the images directly to tensor format for future uses. Due to the large amount of data, the code will take forever to run. So we modified the code and used only one folder of the data(the 'highway' folder) to make future code run efficiently.

```
def load_data():
  data_list = []
  folder_dir = 'data/highway'
  file_dirlist = os.listdir(folder_dir)
  for file_dir in file_dirlist:
    split = os.path.splitext(file_dir)
    if split[1] == '.jpg' or split[1] == '.png' or split[1] == '.jpeg':
      file_full_path = os.path.join(folder_dir, file_dir)
      image_tensor = torchvision.io.read_image(file_full_path)
      data_list.append(image_tensor.div(256))
  return data_list
```

4. We utilized the "create_traintest_set" function to create train sets and test sets, based on the criteria that 80% of all the data are considered as train set and 20% of all the data are considered as test set.
5. By defining the "separate_cover_secret" function, we split the cover set from the secret set. We set the first half of the data as a cover set and the second half of the data as a secret set for both train set and test set.

```
[ ] def separate_cover_secret(train_set, test_set):
      train_set_cover = train_set[int(len(train_set) / 2):]
      train_set_secret = train_set[:int(len(train_set) / 2)]

      test_set_cover = test_set[int(len(test_set) / 2):]
      test_set_secret = test_set[:int(len(test_set) / 2)]
      return train_set_cover, train_set_secret, test_set_cover, test_set_secret
```

6. We created a train model based on an example code in github listed under the description of the second task (https://github.com/fpingham/DeepSteg/blob/master/DeepSteganography.ipynb). And we modified the code to match the situation of our data. Because there were some differences between the data dimension of our tensor data and the data dimension of the build network, we adjusted its dimension by employing ‘unsqueeze’. Also, we met an error regarding a different format of tensor. We use “float()” to unify the format to fix this error. Since we separated the cover set and secret set in the previous step, we deleted the lines for creating cover set and secret set originally in the example code, input the cover set and secret set directly when run the ‘train_model’ function.

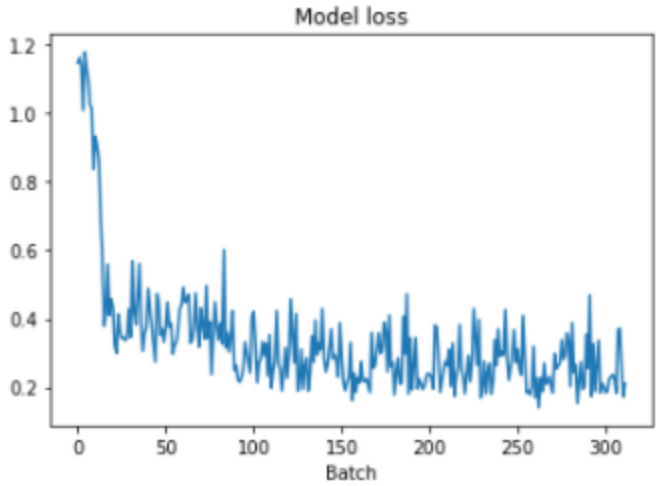
*: The function torch.nn.conv2d() is the main function of our model. From the PyTorch website we got following equation: (https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html)

In the simplest case, the output value of the layer with input size (N, C_{in}, H, W) and output $(N, C_{\text{out}}, H_{\text{out}}, W_{\text{out}})$ can be precisely described as:

$$\text{out}(N_i, C_{\text{out}_j}) = \text{bias}(C_{\text{out}_j}) + \sum_{k=0}^{C_{\text{in}}-1} \text{weight}(C_{\text{out}_j}, k) \star \text{input}(N_i, k)$$

Then we are training the values in weight to minimize the optimizer function.

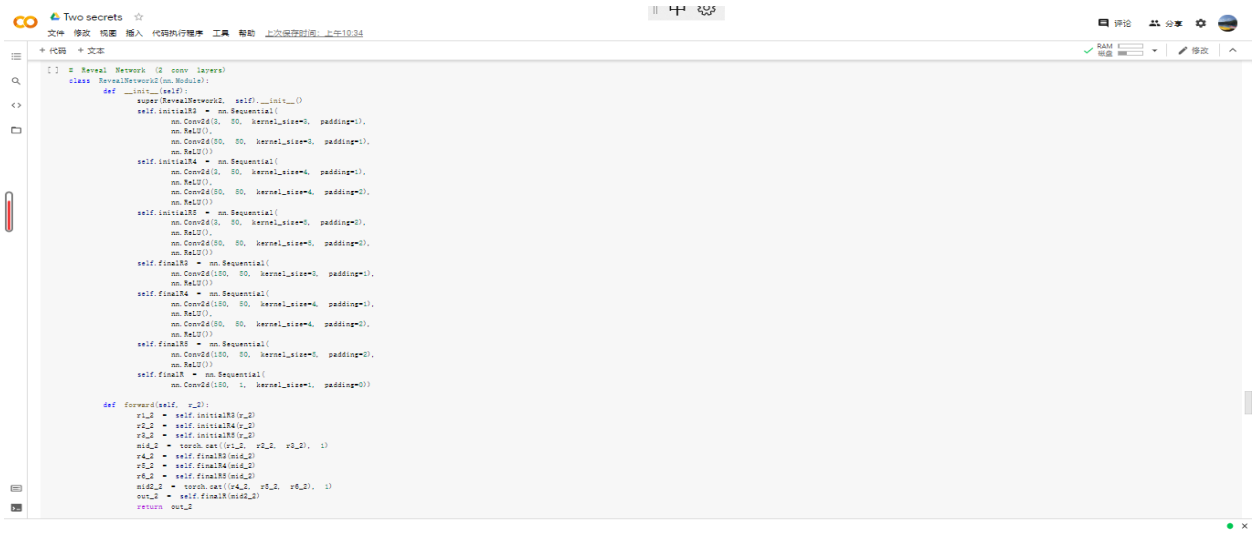
Results:



We constructed the structure of our code based on the code provided in the github link and then modified the code to match the situation of our data. In contrast with task 1, we spent more time solving task 2 and we encountered more questions during the process of task 2. When we ran the model, the colab showed errors many times. For example, it showed that the format of the original code does not match the format of our code and we also encountered the problem regarding the loss of one dimension. In order to solve these issues we encountered, we searched for error messages online and read the reply of some scholar. From the documentation provided by the scholars, we learned a lot about attributes and the definition of some parameter. Based on the article and important information, we modify our code and then get results successfully.

Optional Task (merge two secret images in one cover image):

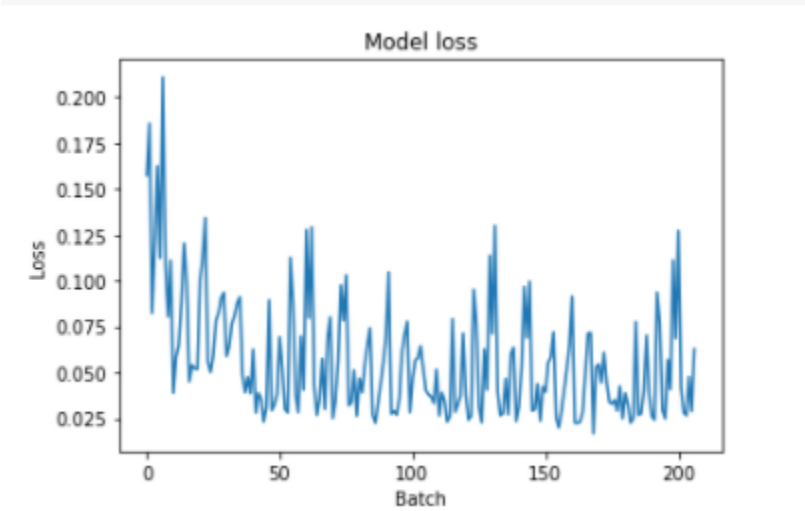
1. Main Strategy: The main strategy of the optional task is hiding two secret images in one cover image and setting two classes to reveal both of them.
2. At first, we set a similar class for all three of PrepNetwork, HidingNetwork, and RevealNetwork classes to deal with the second secret image.



One of the twin networks

3. Then we train the Net() network in a similar way as task 2.
4. We got a good result on this.

Results:



*: LSB should be able to deal with 3 or 4, or maybe 8 images by simply distributing the eight digits to images. Of course, the more secret images, the greater loss since less digits are distributed to a single image. In comparison, NN does not have an image number limitation.

However, with the increase in the number of secret images, the time consumed to train the model increases exponentially.

Comparison: LSB and NN

1. NN preserves much more information than LSB does. LSB only preserves half information of the original image while NN preserves almost all information of the original image, though the information preserved by NN is not exactly accurate.
2. NN is way more time consuming than LSB.