Steganography with LSB and Neural Network

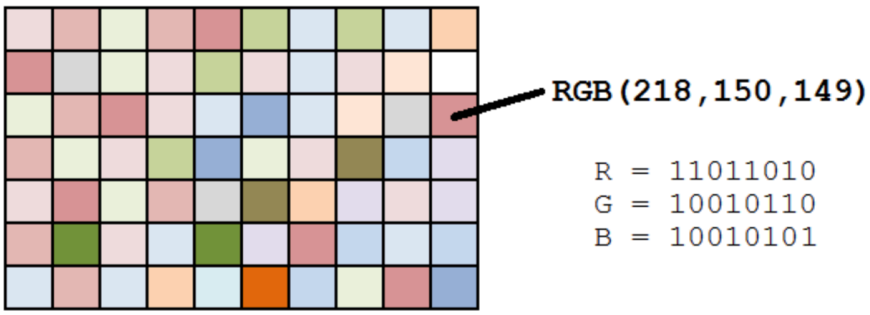
Meng Xu, Jiaxiang Peng

In this project, we examine LSB (Least Significant Bits) and NN (Neural Network) to encrypt and decrypt images. Our goal is to merge the secret image to the original image to get the encrypted image and then pull the secret image out of the encrypted image to get the decrypted image. In this process, we want the encrypted image to be as similar to the original image as possible and the encrypted image to be as similar to the secret image as possible.

Mathematically: let x be the original image, y the secret image, $z = E(x, y)$ the encrypted image, $D(z)$ the decrypted image. We will try to minimize $\| z - x \|$ with the constraint that $\| D(z) - y \|$ should also be as small as possible, which means that our objective function is $\min_{E,D} \| x - E(x, y) \| + \gamma \| y - D(z) \|$. γ is a parameter chosen at our choice used in the neural network approach.

Task 1:

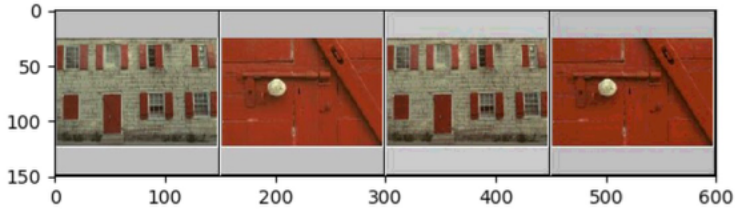
RBG color model: WLOG, here we examine with 256*256 pixels digital images. Each image is composed of 256*256 pixels. Each pixel is composed of 3 values (3 channels): R (red), G (green), B (blue). The value of each channel is represented by an integer ranging from 0~255. And this integer is represented by an 8-bit binary string. For example:

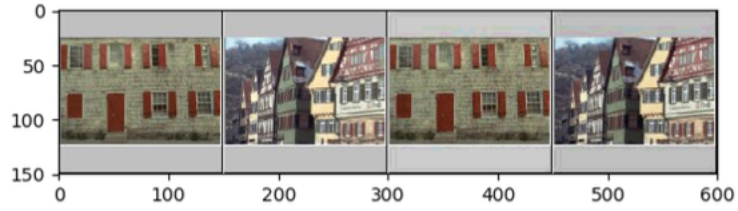


LSB algorithm: for each channel in each pixel of the original image, we replace the least important four digits with the most important four digits of the corresponding channel in corresponding pixel of the secret image. Since we replaced the least important info of the original with the most important info of the secret, the encrypted image will be close to the original, and simultaneously the decrypted image is similar to the secret image.

Pros and cons: LSB is relatively simple to implement and low-cost. But since this algorithm is public and simple, it's easy for others to detect the secret image from our encrypted image, then it lost its purpose. And since it alters the last bit, it may distory the statistics of the last bit, which may obey some heuristic distribution.

Implementation & Result: In the ***LSB → Sample Code*** folder, you can find our implementation of LSB which adapted from https://github.com/kelvins/steganography. And we sampled some pictures from the Kodak images file given. The test results (folder of pictures) can be found in ***LSB → Test Result***. From our perspective, the results are already satisfying, the encrypted and decrypted images look good.





Task 2

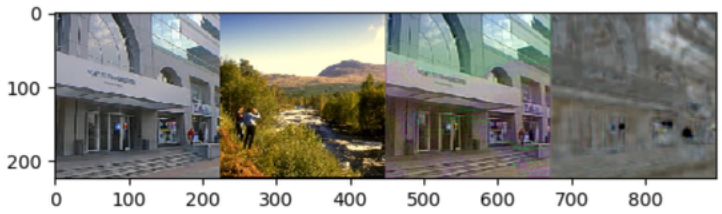
Algorithm:

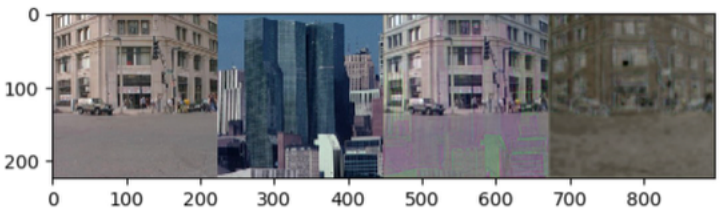
1. Data preparing: we convert every pixel value from 0~255 to binary strings for convenience of searching when we train the model later.
2. Extraction of features of secret image: extract the important features of secret image for encryption in original image later. Use 5 layers of convolutional neural networks with 3x3, 4x4, 5x5 patches. It's better to use multiple convolution sizes so that we can extract more important features.
3. Encryption function E: Use the info extracted from the secret image to concatenate with info of the channels in the original image. Also use 3x3, 4x4, 5x5 patches convolution.
4. Decryption function D: Add random noise to the result of step 3 and use the same convolution to get decrypted image. Adding random noise in the training process adds fault-tolerance.
5. Calculate the loss between original and encrypted images and the loss between the the secret and decrypted images and get the weighted sum. We used $\gamma = 0.75$ and users can adjust this ratio.

Pros and cons: Theoretically, if we have big enough datasets, good enough device (eg. Gpu) and enough time, NN should produce very good model to encrypt and decrypt. But in order to get good results, the number of parameters should be 100k or 1000k. Need to train different models if we deal with different sizes of images, or we can solve this problem by data preparing. If we have more than one secret image, we have to modify the model a lot.

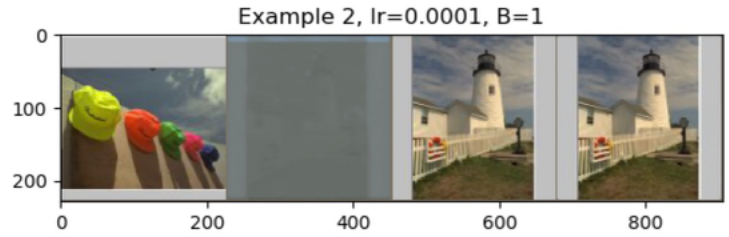
Implementation & Result: We implemented two sample codes which adapted from https://papers.nips.cc/paper/6802-hiding-images-in-plain-sight-deep-steganography.pdf and https://github.com/alexandremuzio/deep-steg. You can find the two different codes (with trained models) in ***Neural Network → Approach 1→ Code*** and ***Neural Network → Approach 2 → Code*** respectively. In each implementation, we used all the 256*256 images in the Kaggle dataset provided and split the data into 80% train and 20% validation. Since the training processes are extremely time-consuming, we provided the trained model using these data, which can be found in ***Neural Network → Approach 1 → Trained Model*** and ***Neural Network → Approach 2 → Trained Model***, respectively. And again we test the models with samples from the Kodak images file given, the image results are in ***Neural Network → Approach 1 → Test Result*** and ***Neural Network → Approach 1 → Test Result***, respectively. Due to the limited time frame and device (Macbook), the results of these two approaches are both reeeeeeally bad. Theoretically, we should attain better results using NN than using LSB.

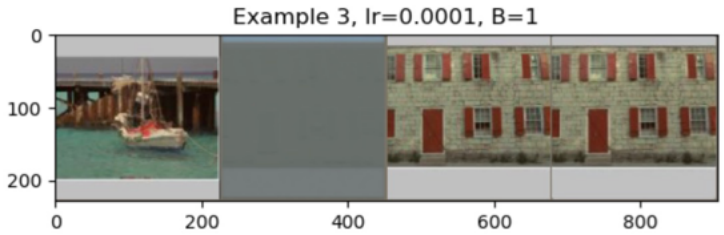
Approach 1:





Approach 2:





Optional Task

If we have two secret images, for NN to work, we can train new model so that it accepts two secret images. Or we can use NN to encrypt secret image 1 in secret image 2, then encrypt the merged secret image in the original image (use NN twice).

If we use LSB, we can try put two 4-bit important info from the secret images into the original. Or similar to NN, we can use LSB to encrypt secret image 1 into secret 2 and then encrypt into the original (use LSB twice).

Another way could be using DCT (discrete-cosine-transform). Replace the high-frequency features of the original with the low-frequency features of the secret images because human eyes are sensitive to low-frequency info and insensitive to high-frequency info.