

Compression with Constraints: Steganography

Haodong Hu

September 15, 2021

1 Introduction

The focus of this project will be **Compression with Constraints: Steganography**. **Steganography**[7] is the practice of concealing a message within another message or a physical object. In computing/electronic contexts, a computer file, message, image, or video is concealed within another file, message, image, or video[5]. According to online resources, the first recorded use of the term was in 1499 by Johannes Trithemius in his *Steganographia*, a treatise on cryptography and steganography, disguised as a book on magic. Generally, the hidden messages appear to be (or to be part of) something else: images, articles, shopping lists, or some other cover text. For example, the hidden message may be in invisible ink between the visible lines of a private letter. Some implementations of steganography that lack a shared secret are forms of security through obscurity, and key-dependent steganographic schemes adhere to Kerckhoffs's principle[12].

The advantage of steganography over cryptography alone is that the intended secret message does not attract attention to itself as an object of scrutiny[6]. Plainly visible encrypted messages, no matter how unbreakable they are, arouse interest and may in themselves be incriminating in countries in which encryption is illegal. Whereas cryptography is the practice of protecting the contents of a message alone, steganography is concerned both with concealing the fact that a secret message is being sent and its contents. Steganography includes the concealment of information within computer files. In digital steganography, electronic communications may include steganographic coding inside of a transport layer, such as a document file, image file, program or protocol. Media files are ideal for steganographic transmission because of their large size. For example, a sender might start with an innocuous image file and adjust the color of every hundredth pixel to correspond to a letter in the alphabet. The change is so subtle that someone who is not specifically looking for it is unlikely to notice the change[15].

In this project, we will implement imaging processing libraries and various optimization techniques in order to finish each given task and constantly reflect on how to detect steganographic images without the original images. In the following sections, we will first introduce the basic mathematical concepts of steganography. Then, we will showcase our coding process, including the implementation of **least significant bits** and approximation of parameters through **neural network**, for both task 1 and task 2.

2 Steganography

In this section, we will first introduce different types of steganography as well as the areas where we could implement each of them. Then, we will explain the model of steganography, which can help us understand how the computing machine works under various circumstances. Lastly, we will show the basic mathematical concepts of steganography and how we can transform different mathematical equations into computing algorithms and utilize them for our tasks.

2.1 Types of Steganography

As we have explained in Section 1, steganography is the method of hiding secret data in any image, audio, or video. In other words, the main purpose of steganography is to hide the intended information within any image, audio, or video that doesn't appear to be secret just by looking at it. The idea behind image-based steganography is very simple. Images are composed of digital data (pixels), which describes what's

inside the picture, usually the colors of all the pixels. Since we know every image is made up of pixels and every pixel contains 3-values (red, green, blue)[13]. As computing technology evolves rapidly and machine algorithms grow more sophisticated, the number of types of steganography has also been increasing and by now, we have text steganography, image steganography, video steganography, network steganography, email steganography, and audio steganography[11].

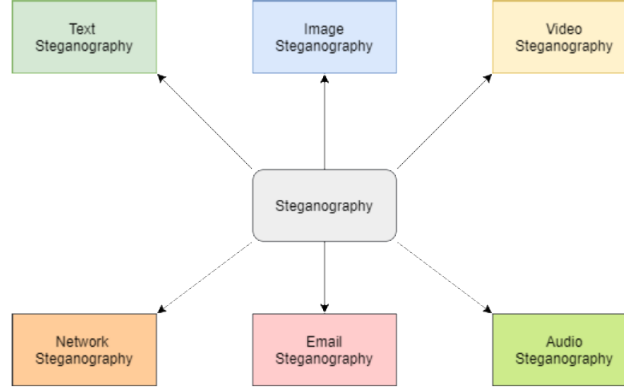


Figure 1: Steganography works have been carried out on different transmission media like images, video, text, or audio

2.2 Steganographic Model[14]

As seen in the below graph, both the original image file(X) and secret message (M) that needs to be hidden are fed into a steganographic encoder as input. Steganographic Encoder function, $f(X, M, K)$ embeds the secret message into a cover image file by using techniques like least significant bit encoding. The resulting stego image looks very similar to your cover image file, with no visible changes. This completes encoding. To retrieve the secret message, stego object is fed into Steganographic Decoder[3].

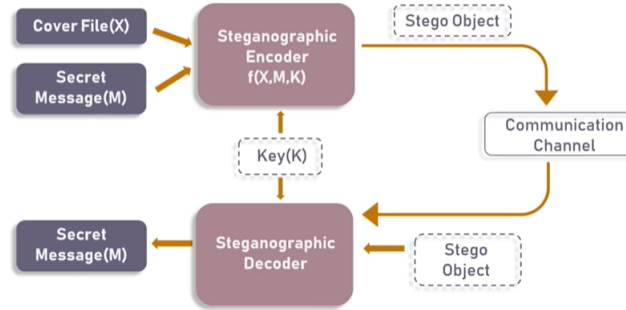


Figure 2: Image source: <https://www.edureka.co/blog/steganography-tutorial>

2.3 Basic Mathematical Concepts

The steganography has two important components: **Encryption** and **Decryption**[9]. For example, assume that we have an original image and secret image, then encryption in this case means that as a sender, our task is to make the original image and secret image merge into one single image. There are two main purposes behind it. First of all, our outcome must be an image, if this image stays far away from the original image, then it will attract other people's attention, which will count as a failure. Moreover, our outcome image must also convey the information of the secret image, and since any blending of information will change each other, the task itself will become more complicated. After the encryption part, our outcome image will have

two parts of information coming from original image and the secret image. As the receiver, our task now will be inverting the encryption process in order to recover the secret image as much as possible, and this is decryption. Notice that such process can be represented mathematically: let x be the original image and y be the secret image, then $z = E(x, y)$ is the encrypted image and E is the encryption function[16]. Now, the task becomes to minimize

$$||z - x||$$

Above minimization problem will subject to another constraint, which is the decryption function D can recover sufficient information of the secrete image, and this can be expressed as

$$||D(z) - y||$$

which should be as small as possible[2]. It is also possible for us to construct a unified objective function which can be represented as

$$\min_{E,D} ||x - E(x, y)|| + \gamma ||y - D(z)||$$

where E and D are the parameters that need to be obtained and γ is the parameter that depends on the choice of the user.

Remark 1. There are other constraints from the images. Since images are pixels, each pixel contains 3 channels, which are R (Red), G (Green), B (Blue), and each one is a 8-bit integer that goes from 0 to 255. If the image has 4 channels RGBA, then it will provide more information. Therefore, above optimization problem also has constraints that $E(x, y)$ and $D(z)$ must be images.

3 Algorithms of Least Significant Bits

In this section, we will first introduce the basic concepts of least significant bit. Then, we will briefly discuss the ideas of most significant bit and compare the differences between least significant bit and most significant bit. In later subsections, we will talk about the code implemented for least significant bit (including the required Python toolkit for tasks) as well as the results that we have obtained for each task.

3.1 Basic Concepts

In computing, the **least significant bit**[4] is the bit which is farthest to the right and holds the least value in a multi-bit binary number. As binary numbers are largely used in computing and other related areas, the least significant bit holds importance, especially when it comes to transmission of binary numbers. Digital data is computed in binary format, and similarly to numerical notation, the right digit is considered the lowest digit whereas the leftmost is considered the highest digit. Due to the positional notation, the least significant bit is also known as the rightmost bit. It is the opposite of the most significant bit, which carries the highest value in a multiple-bit binary number as well as the number which is farthest to the right. In a multi-bit binary number, the significance of a bit decreases as it approaches the least significant bit. Since it is binary, the most significant bit can be either 1 or 0. When a transmission of binary data is done with the least significant bit first technique, the least significant bit is the one which is transmitted first, followed by other bits of increasing significance. The least significant bit is frequently employed in hash functions, checksums and pseudorandom number generators.

3.2 Least Significant Bit or Most Significant Bit

In computer science and information theory, a bit is the smallest possible meaningful piece of information. It is most often expressed as a digit of the binary numeral system: either 0 or 1. A string of 8 bits is called a byte. For example, If we take the binary number 11100111 (231 in decimal), and send it as a string of data of a network, we can send it in two ways: starting from left to right, or starting right to left. These two orderings are commonly called Most Significant Bit First, and Least Significant Bit First, respectively. In this case, we refer to the first, or left-most bit as the **Most Significant Bit** (MSB for short)[10]. The MSB is the bit in a binary sequence that carries the greatest numerical value. For simpler reference, if we

take a look at the equivalent decimal number, 231, the most significant digit is the leading 2. Compared to the other two digits, the leading 2 determines the greatest part of the number's numerical value, as it signifies the hundreds in the number. Analogous to this, the leading 1 in our binary number is its most significant bit. The MSB can also be used to denote the sign bit of a binary number in its one's or two's complement notation, with 1 meaning it's a negative number, and 0 meaning it's a positive number. One's complement of a binary number is obtained by simply inverting all of its digits. However, in order to avoid the signed zero problem (where two possible zeros can exist: +0 and -0, which in turn leads to a number of different problems), two's complement is used. In order to calculate a binary number's two's complement, we first determine its one's complement (by swapping its digits like before), and then adding 1 to it. In basic calculus, this two's complement will behave like the negative number of the starting bit sequence, and it enables the use of adders (digital circuits used for addition) for subtraction. For example: we wish to subtract 1101 from 100111[10]. First, we expand the number so it has the same amount of digits as the minuend. We get 001101. Then we invert its digits to get 110010, and add 1 to it, getting 110011. We then proceed to add the two numbers. $100111 + 110011 = 1011010$. Since we were working with 6-digit numbers, we remove the leading digit from the 7-digit result. This gives us the final result of 11010, or 26 in decimal. In the decimal system, our original task was to subtract 13 from 39, so we can easily verify the result to be accurate.

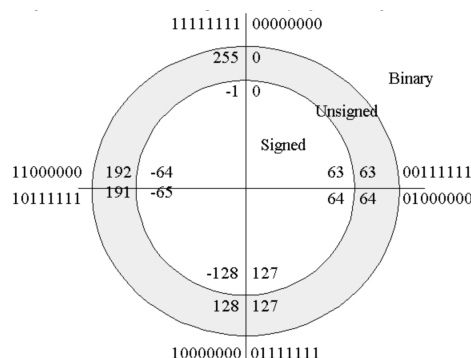


Figure 3: The diagram shows the relationship between binary, signed and unsigned numbers for 8 bit two's complement

The **Least Significant Bit**, also known as LSB, is the right-most bit in a string. It is called that because it has the least effect on the value of the binary number, in the same way as the unit digit in a decimal number has the least effect on the number's value. The LSB also determines whether the given number is odd or even. The number 11100111 is an odd number, since its LSB (1) is an odd number. If we use the term least significant bits (plural), we are commonly referring to the several bits closest to, and including, the LSB. Another property of the least significant bits is that they often change drastically if the number changes. For example, if we add 1 to our example number, 11100111, we will get 11101000[4]. The result of this minimal addition is that the four least significant bits have changed their value. There are a lot of implementations of LSB and some of the most important uses for LSB are hash functions and checksums. A hash function is a function that can be used to assign an index value to digital data, enabling faster look-ups of items in a database or a large table. A checksum is a small piece of data that's used in the verification process for packets sent over a network. For example, every ASCII table character is encoded by a 7-bit binary number, but is stored as a whole byte. The last bit is used as a parity check bit. For example, the letter "A" is encoded as 1000001. A parity check is performed by adding together all the digits in the number. If the digit we get is even, an extra 0 is added at the end, and our letter "A" is stored as 10000010. The letter "C" (1000011) is stored as 10000111, due to the sum of digits being an odd number. This means that the sum of every byte will be an even number, and if, during a transmission, the data gets corrupted or altered, and the sum becomes an odd number, that byte of data is incorrect. There are advantages and disadvantages to every form of error detection. The advantage of the parity bit is that it is not a big block data and is relatively easy to implement, however, it can only be used to detect an odd number of errors,

because an even number of altered bits will cancel each other out in respect to the parity bit.

3.3 Required Toolkit for LSB

Before getting into the formal introduction of the code for task 1, it is important to first acknowledge the required Python packages and toolkit for the algorithms. For task 1, some of our code is based on the work from source: <https://github.com/kelvins/steganography>, and therefore, according to this source, the required toolkit for the algorithm is **PIL**, and below is the code from Python which will import the toolkit directly.

```
from PIL import Image
```

Figure 4: PIL is the Python Imaging Library which provides the python interpreter with image editing capabilities

Moreover, **PIL.Image.new()** method creates a new image with the given mode and size. Size is given as a (width, height)-tuple, in pixels. The color is given as a single value for single-band images, and a tuple for multi-band images (with one value for each band). Also notice that one can create any types of color as they want by setting the parameters for “RGB” through this package.

3.4 Algorithms Implementation

We first begin by defining several classes in Python.

```
class Steganography(object):

    @staticmethod
    def __int_to_bin(rgb):
        """Convert an integer tuple to a binary (string) tuple.
        :param rgb: An integer tuple (e.g. (220, 110, 96))
        :return: A string tuple (e.g. ("00101010", "11101011", "00010110"))
        """
        r, g, b = rgb
        return ('{0:08b}'.format(r),
                '{0:08b}'.format(g),
                '{0:08b}'.format(b))
```

Figure 5: Class converts an integer tuple to a binary (string) tuple

```
@staticmethod
def __bin_to_int(rgb):
    """Convert a binary (string) tuple to an integer tuple.
    :param rgb: A string tuple (e.g. ("00101010", "11101011", "00010110"))
    :return: Return an int tuple (e.g. (220, 110, 96))
    """
    r, g, b = rgb
    return (int(r, 2),
            int(g, 2),
            int(b, 2))
```

Figure 6: Class converts a binary (string) tuple to an integer tuple

```

@staticmethod
def __merge_rgb(rgb1, rgb2):
    """Merge two RGB tuples.
    :param rgb1: A string tuple (e.g. ("00101010", "11101011", "00010110"))
    :param rgb2: Another string tuple
    (e.g. ("00101010", "11101011", "00010110"))
    :return: An integer tuple with the two RGB values merged.
    """
    r1, g1, b1 = rgb1
    r2, g2, b2 = rgb2
    rgb = (r1[:4] + r2[:4],
           g1[:4] + g2[:4],
           b1[:4] + b2[:4])
    return rgb

```

Figure 7: Class merges two RGB tuples

```

@staticmethod
def merge(img1, img2):
    """Merge two images. The second one will be merged into the first one.
    :param img1: First image
    :param img2: Second image
    :return: A new merged image.
    """

```

Figure 8: Class merges two images and the second one will be merged into the first one

Then, we check the dimensions of the images by comparing input images and using the command **if**, and since we are inputting 2 images with corresponding numbers, the message “Image 2 should not be larger than Image 1!” will show if the size requirement is not satisfied. After comparing the sizes of input images, we will get the pixel map of the two images by using the code “pixel_map1=img1.load(), pixel_map2=img2.load()”. Then, we will create a new image that will be returned when the code by defining “new_image” and “pixels_new”. Now, we set a black pixel as our default choice and then check if the pixel map position is valid for the second image. If all of the requirements are satisfied, we merge the two pixels and convert it to a integer tuple. Next, we define another class.

```

@staticmethod
def unmerge(img):
    """Unmerge an image.
    :param img: The input image.
    :return: The unmerged/extracted image.
    """

```

Figure 9: Class unmerges an image

After defining the class, we load the pixel map first, and then we create the new image and load the pixel map again, and while we do these, we use Tuple to store the image original size and get the RGB (as a string tuple) from the current pixel. Then, we extract the last 4 bits (corresponding to the hidden image) and concatenate 4 zero bits because we are working with 8 bit. Next, we convert it to an integer tuple and if the position is valid, we will store it as the last valid position. Finally, we crop the image based on the ‘valid’ pixels and return “new_image”. Below are the screenshots of the code implemented after defining the classes in Python.

```

# Check the images dimensions
if img2.size[0] > img1.size[0] or img2.size[1] > img1.size[1]:
    raise ValueError('Image 2 should not be larger than Image 1!')

# Get the pixel map of the two images
pixel_map1 = img1.load()
pixel_map2 = img2.load()

# Create a new image that will be outputted
new_image = Image.new(img1.mode, img1.size)
pixels_new = new_image.load()

for i in range(img1.size[0]):
    for j in range(img1.size[1]):
        rgb1 = Steganography.__int_to_bin(pixel_map1[i, j])

        # Use a black pixel as default
        rgb2 = Steganography.__int_to_bin((0, 0, 0))

        # Check if the pixel map position is valid for the second image
        if i < img2.size[0] and j < img2.size[1]:
            rgb2 = Steganography.__int_to_bin(pixel_map2[i, j])

        # Merge the two pixels and convert it to a integer tuple
        rgb = Steganography.__merge_rgb(rgb1, rgb2)

        pixels_new[i, j] = Steganography.__bin_to_int(rgb)

return new_image

```

Figure 10: Code explanations are embedded within Python and also explained in the report

```

# Load the pixel map
pixel_map = img.load()

# Create the new image and load the pixel map
new_image = Image.new(img.mode, img.size)
pixels_new = new_image.load()

# Tuple used to store the image original size
original_size = img.size

for i in range(img.size[0]):
    for j in range(img.size[1]):
        # Get the RGB (as a string tuple) from the current pixel
        r, g, b = Steganography.__int_to_bin(pixel_map[i, j])

        # Extract the last 4 bits (corresponding to the hidden image)
        # Concatenate 4 zero bits because we are working with 8 bit
        rgb = (r[4:] + '0000',
              g[4:] + '0000',
              b[4:] + '0000')

        # Convert it to an integer tuple
        pixels_new[i, j] = Steganography.__bin_to_int(rgb)

        # If this is a 'valid' position, store it
        # as the last valid position
        if pixels_new[i, j] != (0, 0, 0):
            original_size = (i + 1, j + 1)

# Crop the image based on the 'valid' pixels
new_image = new_image.crop((0, 0, original_size[0], original_size[1]))

return new_image

#@title
def merge(img1, img2, output):
    merged_image = Steganography.merge(Image.open(img1), Image.open(img2))
    merged_image.save(output)

def unmerge(img, output):
    unmerged_image = Steganography.unmerge(Image.open(img))
    unmerged_image.save(output)

```

Figure 11: Code explanations are embedded within Python and also explained in the report

4 Neural Network for Hidden Layer Image

In this section, we will first discuss the background and basic concepts of Neural Network[1]. We will also talk about the mathematical strategies that are helpful in terms of training an neural network. Then, we will implement various strategies for task 2 through Python code and give detailed explanations of each part of the code. Finally, we will show part of the results of our code (since the results are too long to be presented in the report) and discuss the overall performance of the algorithms.

4.1 Neural Network Training

Once a network has been structured for a particular application, that network is ready to be trained. To start this process the initial weights are chosen randomly. Then, the training, or learning, begins. There are two approaches to training: **supervised** and **unsupervised**. Supervised training involves a mechanism of providing the network with the desired output either by manually "grading" the network's performance or by providing the desired outputs with the inputs. Unsupervised training is where the network has to make sense of the inputs without outside help. In supervised training, both the inputs and the outputs are provided. The network then processes the inputs and compares its resulting outputs against the desired outputs[8]. Errors are then propagated back through the system, causing the system to adjust the weights which control the network. This process occurs over and over as the weights are continually tweaked. The set of data which enables the training is called the "training set." During the training of a network the same set of data is processed many times as the connection weights are ever refined. The other type of training is called unsupervised training. In unsupervised training, the network is provided with inputs but not with desired outputs. The system itself must then decide what features it will use to group the input data. This is often referred to as self-organization or adaption[1]. At the present time, unsupervised learning is not well understood. This adaption to the environment is the promise which would enable science fiction types of robots to continually learn on their own as they encounter new situations and new environments. Life is filled with situations where exact training sets do not exist. Some of these situations involve military action where new combat techniques and new weapons might be encountered. Because of this unexpected aspect to life and the human desire to be prepared, there continues to be research into, and hope for, this field. Yet, at the present time, the vast bulk of neural network work is in systems with supervised learning. Supervised learning is achieving results.

However, training could be very hard and complicated. When we start off with our neural network, we initialize our weights randomly, which will not give us very good results. In the process of training, we want to start with a bad performing neural network and wind up with network with high accuracy. In terms of loss function, we want our loss function to much lower in the end of training. Improving the network is possible, because we can change its function by adjusting weights. We want to find another function that performs better than the initial one. Also, the weights of a neural network with hidden layers are highly interdependent. For example, consider the highlighted connection in the first layer of the three layer network below. If we tweak the weight on that connection slightly, it will impact not only the neuron it propagates to directly, but also all of the neurons in the next two layers as well, and thus affect all the outputs.

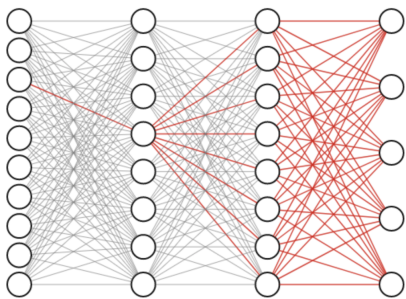


Figure 12: Tweaking the weight of one connection in the first layer will affect just one neuron in the next layer, but because of fully-connection, all neurons in subsequent layers will be changed

There are a lot of algorithms that optimize functions. These algorithms can gradient-based or not, in sense that they are not only using the information provided by the function, but also by its gradient. In the following sections, we will list the required toolkit of neural network for our algorithms and then there will be detailed explanations about each part of our code.

4.2 Required Toolkit for Neural Network

- **io**: allows us to manage the file-related input and output operations. The advantage of using the io module is that the classes and functions available allows us to extend the functionality to enable writing to the Unicode data.
- **os**: provides functions for interacting with the operating system. os comes under Python's standard utility modules. This module provides a portable way of using operating system-dependent functionality. The "os" and "os.path" modules include many functions to interact with the file system.
- **PIL**: adds image processing capabilities to Python interpreter and provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities.
- **numpy**: the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and more.
- **matplotlib.pyplot**: a collection of functions that make matplotlib work like MATLAB. Each "pyplot" function makes some change to a figure: such as creating a figure, creating a plotting area in a figure, plotting some lines in a plotting area and decorating the plot with labels.
- **IPython**: a command shell for interactive computing in multiple programming languages, originally developed for the Python programming language, that offers introspection, rich media, shell syntax, tab completion, and history.
- **scipy**: an open-source library used for solving mathematical, scientific, engineering, and technical problems. It allows users to manipulate the data and visualize the data using a wide range of high-level Python commands. SciPy is built on the Python NumPy extension.
- **itertools**: a module in python and it is used to iterate over data structures that can be stepped over using a for-loop. Such data structures are also known as iterables. This module incorporates functions that utilize computational resources efficiently. Using this module also tends to enhance the readability and maintainability of the code.
- **PyTorch**: a library for Python programs that facilitates building deep learning projects. PyTorch emphasizes flexibility and allows deep learning models to be expressed in idiomatic Python.
- **torch.autograd**: provides classes and functions implementing automatic differentiation of arbitrary scalar valued functions.
- **torch.nn**: provides many more classes and modules to implement and train the neural network.
- **torch.nn.optim**: a package implementing various optimization algorithms. Most commonly used methods are already supported, and the interface is general enough, so that more sophisticated ones can be also easily integrated in the future.
- **pickle**: is primarily used in serializing and deserializing a Python object structure. In other words, it's the process of converting a Python object into a byte stream to store it in a file/database, maintain program state across sessions, or transport data over the network. The pickled byte stream can be used to re-create the original object hierarchy by unpickling the stream.
- **torchvision**: a library for Computer Vision that goes hand in hand with PyTorch. It has utilities for efficient Image and Video transformations, some commonly used pre-trained models, and some datasets.
- **torchvision.transforms**: are common image transformations. They can be chained together using "compose". Most transform classes have a function equivalent: functional transforms give fine-grained control over the transformations. This is useful if you have to build a more complex transformation pipeline.

- **random:** an inbuilt function of the random module in Python3. The random module gives access to various useful functions and one of them being able to generate random floating numbers.

Below is a screenshot of the packages and toolkit mentioned above, and the screenshot also demonstrates how each package and toolkit is installed and imported.

```
import io
import os
from PIL import Image
import numpy as np
import matplotlib.pyplot as plt
import IPython
from scipy import fftpack
from itertools import islice
import matplotlib.pyplot as plt
import numpy as np
import torch
from torch.autograd import Variable
from torch import utils
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import pickle
from torchvision import datasets, utils
from torch.utils.data import DataLoader, Dataset
import torchvision.transforms as transforms
from torchvision.transforms import ToPILImage
from random import shuffle
```

Figure 13: The function and purpose of each toolkit and package are explained above

4.3 Algorithms Implementation for Task 2

First, we set up the hyper parameter. In order to do this, we set up the number of learning cycles and batch size for the model learning (learning rate). Then, the path for the data and the model save is set, and we check if the GPU allows training the model and folder for the model exists or not. If not, we will create one.

```
# Hyper Parameters
num_epochs = 110
batch_size = 64
learning_rate = 0.001
beta = 1
data_path = "data"
std = [0.229, 0.224, 0.225]
mean = [0.485, 0.456, 0.406]
checkpoints_path = "models"
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
if not os.path.exists(checkpoints_path): os.mkdir(checkpoints_path)
```

Figure 14: Some algorithms are based on the source: <http://cs231n.stanford.edu/reports/2017/pdfs/101.pdf>

Then, we continue to define the loss function. “loss_over” is the standard MSE between the actual cover image and the created cover image. “loss_secret” is the MSE between the actual encrypted image and the created decrypted image. “loss_all” is a hyper parameter that is used to control the information that needs to rebuild, and we will also add a gaussian noise function.

```

def customized_loss(S_prime, C_prime, S, C, B):
    ''' Calculates loss specified on the paper.'''

    loss_cover = torch.nn.functional.mse_loss(C_prime, C)
    loss_secret = torch.nn.functional.mse_loss(S_prime, S)
    loss_all = loss_cover + B * loss_secret
    return loss_all, loss_cover, loss_secret

def gaussian(tensor, mean=0, stddev=0.1):
    '''Adds random noise to a tensor.'''

    # noise = torch.nn.init.normal(torch.Tensor(tensor.size()), 0, 0.1)
    noise = torch.nn.init.normal_(torch.Tensor(tensor.size()), 0, 0.1).cuda()
    return (tensor + noise).clone().detach().requires_grad_(True)

```

Figure 15: Loss function and gaussian noise function are defined

Next, the “PrepNet” is used to prepare to hide the encrypted image. The main purpose is when the encrypted image (size MM) is smaller than the carrier image (NN), the preliminary network gradually increases the size of the encrypted image to the size of the overlay image, thereby reducing the bits of the confidential image Distributed to the entire NN pixels. Secondly, the more important purpose related to hidden images of all sizes is to convert color-based pixels into more useful features in order to encode images succinctly. Here are mainly three convolution kernels with different sizes of 3×3 , 4×4 , and 5×5 , and each convolutional layer has 50 filters. Since the input images are color pictures, that is, the images are 3-dimensional, so the first ConvNet layer from p1 to p3 has 3 input channels and 50 output, and then the input and output channels of the second layer Both are 50. The “relu” activation function is used behind each layer, and padding is added to keep the w and h of the feature map consistent with the input image w and h. The number of input channels of the first layer from p4 to p6 is 150, and the output is 50, mainly because we spliced the features from p1 to p3 into a feature map with 150 channels, and then the input and output channels of the second layer. Also note that y is the splicing feature maps from p4 to p6 and output.

```

class PrepNet(nn.Module):
    def __init__(self):
        super(PrepNet, self).__init__()

        # Preparation Network
        self.p1 = nn.Sequential(
            nn.Conv2d(3, 50, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=3, padding=1),
            nn.ReLU()
        )

        self.p2 = nn.Sequential(
            nn.Conv2d(3, 50, kernel_size=4, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=4, padding=2),
            nn.ReLU()
        )

        self.p3 = nn.Sequential(
            nn.Conv2d(3, 50, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=5, padding=2),
            nn.ReLU()
        )

        self.p4 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=3, padding=1),
            nn.ReLU()
        )

        self.p5 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=4, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=4, padding=2),
            nn.ReLU()
        )

        self.p6 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=5, padding=2),
            nn.ReLU()
        )

    def forward(self, S):
        p1 = self.p1(S)
        p2 = self.p2(S)
        p3 = self.p3(S)
        x = torch.cat((p1, p2, p3), 1)
        p4 = self.p4(x)
        p5 = self.p5(x)
        p6 = self.p6(x)
        y = torch.cat((p4, p5, p6), 1)
        return y

```

Figure 16: The numbers are all 50, and the “relu” activation function is also used, and padding is added to keep the w and h of the feature map consistent with the input image w and h.

Now, “EncoderNet” is defined in such way that it takes the output of the preparation network and the carrier image as input, and create a container image. Similar to the preparation network, three convolution kernels of 3×4 , 4×4 , and 5×5 are also used here, and each convolutional layer has 50 Filters, but the only difference is that the network combines the output of the preliminary network and the carrier image as input, removes the carrier image, and displays the encrypted image. Moreover, “forward” here means that there are two main differences between the model structure of the coding network and the model structure of the preparation network. The first is that because this takes the output of the preparation network and the carrier image as input at the same time, the first layer of e1 to e3 in this place The number of input channels is 153, and finally, a convolutional layer with a convolution kernel of 1 is used to complete the final embedding, which is to generate an image with hidden information and generate noise with the same Gaussian distribution as the image with hidden information. The mean value is 0, the standard deviation is 0.1.

```
class EncoderNet(nn.Module):
    def __init__(self):
        super(EncoderNet, self).__init__()
        self.e1 = nn.Sequential(
            nn.Conv2d(153, 50, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=3, padding=1),
            nn.ReLU(),
        )
        self.e2 = nn.Sequential(
            nn.Conv2d(153, 50, kernel_size=4, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=4, padding=2),
            nn.ReLU(),
        )
        self.e3 = nn.Sequential(
            nn.Conv2d(153, 50, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=5, padding=2),
            nn.ReLU(),
        )
        self.e4 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=5, padding=2),
            nn.ReLU()
        )
        self.e5 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=5, padding=2),
            nn.ReLU()
        )
        self.e6 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=5, padding=2),
            nn.ReLU()
        )
        self.out = nn.Sequential(
            nn.Conv2d(150, 3, kernel_size=1, padding=0)
        )
    def forward(self, x):
        e1 = self.e1(x)
        e2 = self.e2(x)
        e3 = self.e3(x)
        x = torch.cat((e1, e2, e3), 1)
        e4 = self.e4(x)
        e5 = self.e5(x)
        e6 = self.e6(x)
        x = torch.cat((e4, e5, e6), 1)
        y = self.out(x)
        y_noise = gaussian(y.data, 0, 0.1)
        return y, y_noise
```

Figure 17: Class “EncoderNet” is defined

Then, we define a decoding network, and the decoding network is similar to the encoding network, mainly used to decrypt images.

```

class DecoderNet(nn.Module):
    def __init__(self):
        super(DecoderNet, self).__init__()

        self.d1 = nn.Sequential(
            nn.Conv2d(3, 50, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=3, padding=1),
            nn.ReLU()
        )

        self.d2 = nn.Sequential(
            nn.Conv2d(3, 50, kernel_size=4, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=4, padding=2),
            nn.ReLU()
        )

        self.d3 = nn.Sequential(
            nn.Conv2d(3, 50, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=5, padding=2),
            nn.ReLU()
        )

        self.d4 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=3, padding=1),
            nn.ReLU()
        )

        self.d5 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=4, padding=1),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=4, padding=2),
            nn.ReLU()
        )

        self.d6 = nn.Sequential(
            nn.Conv2d(150, 50, kernel_size=5, padding=2),
            nn.ReLU(),
            nn.Conv2d(50, 50, kernel_size=5, padding=2),
            nn.ReLU()
        )

        self.final = nn.Sequential(
            nn.Conv2d(150, 3, kernel_size=1, padding=0)
        )

    def forward(self, x):
        d1 = self.d1(x)
        d2 = self.d2(x)
        d3 = self.d3(x)
        x = torch.cat((d1, d2, d3), 1)
        d4 = self.d4(x)
        d5 = self.d5(x)
        d6 = self.d6(x)
        x = torch.cat((d4, d5, d6), 1)
        y = self.final(x)
        return y

```

Figure 18: Decoding network is defined similarly as encoding one

Next, we create a model that is the application of the above three Net and embed the model on GPU. Then, we load the picture and keep the path of the file in a list. Lastly, we randomize the previous list. Separate the data into 80% train data, and 20% test data.

```

class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        self.p = PrepNet()
        self.e = EncoderNet()
        self.d = DecoderNet()

    def forward(self, S, C):
        S = self.p(S)
        x = torch.cat((S, C), 1)
        x, x_noise = self.e(x)
        y = self.d(x_noise)
        return x, y

model = Model().cuda()

def load_data(data_path):
    data_list = []
    for root, dirs, files in os.walk(data_path):
        for file in files:
            if file.endswith(".jpg"):
                data_list.append(os.path.join(root, file))
    return data_list

data_list = load_data(data_path)
shuffle(data_list)
data_list = np.array(data_list)
train_data = data_list[0:int(0.8*len(data_list))]
test_data = data_list[int(0.8*len(data_list)):]

```

Figure 19: Preparing the model for neural network

In the next step, we convert the data set, because the size of the image is $256 \times 256 \times 3$, but for computational efficiency, the image is resized to 128, so that more pictures can be loaded at one time.

```

train_transforms = transforms.Compose([
    transforms.Resize(128),
    transforms.RandomCrop(128),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean,
std=std)
])

test_transforms = transforms.Compose([
    transforms.Resize(128),
    transforms.RandomCrop(128),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean,
std=std)
])

class MyDataset(Dataset):

    def __init__(self, imgs, transform=None):
        super(MyDataset, self).__init__()
        self.imgs = imgs
        self.transform = transform

    def __getitem__(self, index):
        img_path = self.imgs[index]
        img = Image.open(img_path)
        if self.transform is not None:
            data = self.transform(img)
        else:
            img = np.asarray(img)
            data = torch.from_numpy(img)
        return data

    def __len__(self):
        '''the length of resuming dataset'''
        return len(self.imgs)

```

Figure 20: Load the picture and save it in the memory

Additionally, here is the way that we create the training set and test set.

```

train_dataset = MyDataset(train_data, train_transforms)
test_dataset = MyDataset(test_data, test_transforms)
#Creates training set
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, num_workers=0,
    pin_memory=True, shuffle=True, drop_last=True)

# Creates test set
test_loader = torch.utils.data.DataLoader(
    test_dataset, batch_size=2, num_workers=0,
    pin_memory=True, shuffle=False, drop_last=True)

```

Figure 21: Creating training and test sets

```

def train_model(train_loader, beta, learning_rate):

    # Save optimizer
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    loss_history = []
    # Iterate over batches performing forward and backward passes
    for epoch in range(num_epochs):

        # Train mode
        model.train()

        train_losses = []
        # Train one epoch
        for idx, train_batch in enumerate(train_loader):

            data = train_batch
            data = data.cuda()

            # Saves secret images and secret covers
            train_covers = data[:len(data)//2]
            train_secrets = data[len(data)//2:]

            # Creates variable from secret and cover images
            train_secrets = train_secrets.clone().detach().requires_grad_(True)
            train_covers = train_covers.clone().detach().requires_grad_(True)

            # Forward + Backward + Optimize
            optimizer.zero_grad()
            train_hidden, train_output = model(train_secrets, train_covers)

            # Calculate loss and perform backprop
            train_loss, train_loss_cover, train_loss_secret = customized_loss(train_output)
            train_loss.backward()
            optimizer.step()

            # Saves training loss
            # print(train_loss.data)
            train_losses.append(torch.Tensor.item(train_loss.data))
            loss_history.append(torch.Tensor.item(train_loss.data))

            # Prints mini-batch losses
            print('Training: Batch {0}/{1}. Loss of {2:.4f}, cover loss of {3:.4f}, sec

        if epoch+1 % 10 == 0:
            torch.save(model.state_dict(), checkpoints_path+'Epoch N().pkl'.format(epoch+1, num_epochs, mean_train_loss))

        mean_train_loss = np.mean(train_losses)

        # Prints epoch average loss
        print ('Epoch {0}/{1}, Average_loss: {2:.4f}'.format(
            epoch+1, num_epochs, mean_train_loss))

    return model, mean_train_loss, loss_history

```

Figure 22: Start training the model. The image here only shows partial code since the original one is too long to be presented

Then, we load the model and check if all keys are matched successfully. Moreover, we define “denormalize” so we can display the content of the image. “imshow” will display the image for us.

```
model, mean_train_loss, loss_history = train_model(train_loader, beta, learning_rate)

Training: Batch 1/33. Loss of 2.6734, cover loss of 1.3899, secret loss of 1.2834
Training: Batch 2/33. Loss of 2.4077, cover loss of 1.1261, secret loss of 1.2816
Training: Batch 3/33. Loss of 7.2546, cover loss of 4.7051, secret loss of 2.5496
Training: Batch 4/33. Loss of 1.9365, cover loss of 0.7241, secret loss of 1.2125
Training: Batch 5/33. Loss of 2.1892, cover loss of 0.8113, secret loss of 1.3779
Training: Batch 6/33. Loss of 2.8949, cover loss of 0.7280, secret loss of 1.3741
Training: Batch 7/33. Loss of 2.0198, cover loss of 0.7026, secret loss of 1.3171
Training: Batch 8/33. Loss of 1.9857, cover loss of 0.7370, secret loss of 1.2486
Training: Batch 9/33. Loss of 1.6941, cover loss of 0.4634, secret loss of 1.2386
Training: Batch 10/33. Loss of 1.8807, cover loss of 0.6040, secret loss of 1.1966
Training: Batch 11/33. Loss of 1.7845, cover loss of 0.4470, secret loss of 1.2575
Training: Batch 12/33. Loss of 1.5844, cover loss of 0.4411, secret loss of 1.0634
Training: Batch 13/33. Loss of 1.4940, cover loss of 0.3229, secret loss of 1.1712
Training: Batch 14/33. Loss of 1.5652, cover loss of 0.3335, secret loss of 1.2317
Training: Batch 15/33. Loss of 1.5183, cover loss of 0.2741, secret loss of 1.2442
Training: Batch 16/33. Loss of 1.2576, cover loss of 0.2166, secret loss of 1.0410
Training: Batch 17/33. Loss of 1.3686, cover loss of 0.2509, secret loss of 1.1177
Training: Batch 18/33. Loss of 1.6837, cover loss of 0.3170, secret loss of 1.2867
Training: Batch 19/33. Loss of 1.4480, cover loss of 0.3071, secret loss of 1.1409
Training: Batch 20/33. Loss of 1.5815, cover loss of 0.2737, secret loss of 1.3077
Training: Batch 21/33. Loss of 1.3449, cover loss of 0.2415, secret loss of 1.1035

model.load_state_dict(torch.load(checkpoints_path+'Epoch N110.pkl'))

<All keys matched successfully>

def denormalize(image, std, mean):
    ''' Denormalizes a tensor of images. '''
    for t in range(3):
        image[t, :, :] = (image[t, :, :] * std[t]) + mean[t]
    return image
def imshow(img, idx, learning_rate, beta):
    '''Prints out an image given in tensor format.'''
    img = denormalize(img, std, mean)
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.title('Example %s lr=%s B=%s' % (idx, learning_rate, beta))
    plt.show()
    return
```

Figure 23: All keys are matched successfully and it will show the image when the code is executed

4.4 Results for Task 2

In the result, we could see the loss on the secret and cover images and the total loss. For every four images, the first one is the secret image that we want to embed into the original one. The second one is the recovered secret images. The third one is the original cover image. The last one is the image that we want to see, the cover image that embedded the secret image. All the content in the data set will be displayed in such form.

```
# Switch to evaluate mode
model.eval()

test_losses = []
# Show images
for idx, test_batch in enumerate(test_loader):
    # Saves images
    data = test_batch.cuda()
    # Saves secret images and secret covers
    test_secret = data[len(data)//2]
    test_cover = data[len(data)//2]

    # Creates variable from secret and cover images
    test_secret = test_secret.clone().detach().requires_grad_(False)
    test_cover = test_cover.clone().detach().requires_grad_(False)
    # Compute output
    test_hidden, test_output = model(test_secret, test_cover)

    # Calculate loss
    test_loss, loss_cover, loss_secret = customized_loss(test_output, test_hidden, test_secret, test_cover)

    test_secret = test_secret.to('cpu')
    test_cover = test_cover.to('cpu')
    test_hidden, test_output = test_hidden.cpu(), test_output.cpu()

    # diff_S, diff_C = np.abs(np.array(test_output.data[0]) - np.array(test_secret.data[0]))
    # print (diff_S, diff_C)
    # test_secret.data, test_output.data, test_cover.data, test_hidden.data, test_output.data
    if idx % 100 == 0:
        print ('Total loss: {:.2f} \nLoss on secret: {:.2f} \nLoss on cover: {:.2f}'.format(test_loss, loss_secret, loss_cover))

        # Creates img tensor
        imgs = [test_secret.data, test_output.data, test_cover.data, test_hidden.data, test_output.data]
        imgs_tsr = torch.cat(imgs, 0)

        # Prints Images
        imshow(utils.make_grid(imgs_tsr), idx+1, learning_rate=learning_rate, beta=beta)

    test_losses.append(torch.Tensor.item(test_loss.data))

mean_test_loss = np.mean(test_losses)
print ('Average loss on test set: {:.2f}'.format(mean_test_loss))

Total loss: 1.20
Loss on secret: 0.81
Loss on cover: 0.00

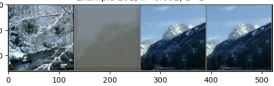
Example 101. lr=0.001, B=1

Total loss: 1.20
Loss on secret: 0.81
Loss on cover: 0.00
```

Figure 24: All of the results for task 2 have been successfully obtained and displayed at the end of the code. Since the results and code themselves are too long to be presented in the report, please refer to original code file for more detailed information

5 Optional Task

First, we will put two secrets into the preparation network to get S1 and S2. Then we will connect a cover with S1 and S2 to construct 156 aisles or roads of data. We will input these data into the encoder network to complete the embedding. At the same time, it will generate two Gaussian distributions of noise. Furthermore, we will input these distributions into the decoder to get two secret data. Since the previously trained model(NN) is not trained for inputting two secrets into one cover. So they are definitely can not be used for doing the option task. The new loss function for training the new model, also for the trained model of putting two s, is shown as

$$f(x, y_1, y_2) = ||x - E(x, y_1, y_2)|| + ||y_2 - D(z_2)|| + ||y_1 - D(z_1)||$$

5.1 Changes Based on the Task 2 Code

We add S1 S2 to get two Gaussian distributions noise. We did not change a lot of codes for it, only for input one more secret. Besides those codes, we need to change all the codes of S to let it be doable for two S.

```
def forward(self, S1, S2, C):
    S1 = self.p1(S1)
    S2 = self.p1(S2)
    x = torch.cat((S1, S2, C), 1)
    x1, x_noise1, x_noise2 = self.e(x)
    y1 = self.d(x_noise1)
    y2 = self.d(x_noise2)
    return x, y1, y2

self.out1 = nn.Sequential(
    nn.Conv2d(150, 3, kernel_size=1, padding=0)
)

self.out2 = nn.Sequential(
    nn.Conv2d(150, 3, kernel_size=1, padding=0)
)

def forward(self, x):
    e1 = self.e1(x)
    e2 = self.e2(x)
    e3 = self.e3(x)
    x = torch.cat((e1, e2, e3), 1)
    e4 = self.e4(x)
    e5 = self.e5(x)
    e6 = self.e6(x)
    x = torch.cat((e4, e5, e6), 1)
    y1 = self.out1(x)
    y2 = self.out2(x)
    y_noise1 = gaussian(y1.data, 0, 0.1)
    y_noise2 = gaussian(y2.data, 0, 0.1)
    return y1, y_noise1, y_noise2
```

Figure 25: Here we have made some changes based on the code from Task 2 so we can fit this problem into the existing model that we have. The figure shows the changes made compared to the original code from Task 2

5.2 Executing Code

However, when we were trying to start training the new mode, the system showed the following error warning: **RuntimeError:** CUDA out of memory. Tried to allocate 20.00 MiB (GPU 0; 11.17 GiB total capacity; 10.56 GiB already allocated; 15.81 MiB free; 10.71 GiB reserved in total by PyTorch). It should be caused by the fact that it has to input two S at the same time and it needs more memory. Therefore, our conclusion is that the NN and SLB can do this option task, but we need to change the logistics of the code and needs a high amount of GPU memory platform to train the model.

6 Conclusion and Future Research

This project establishes a foundational approach to studying steganography by focusing on Compression with Constraints and exploring techniques for concealing and detecting hidden messages in digital media. While significant progress is anticipated through the tasks outlined in this project, several avenues remain for future research and development. These directions aim to address existing challenges, enhance the scope of applications, and further advance the field of steganography.

6.1 Enhanced Detection Techniques Without Original Images

Detecting steganographic modifications without access to the original media is a critical challenge in this domain. Future work could involve the development of advanced machine learning models, particularly deep neural networks, trained on extensive datasets of steganographic and non-steganographic images. Transfer learning and fine-tuning techniques may be employed to improve detection accuracy across different media types, including images, videos, and audio files. Additionally, integrating statistical anomaly detection methods into these models could help identify subtle patterns indicative of hidden messages.

6.2 Optimization of Embedding Algorithms

The efficiency and imperceptibility of embedding steganographic messages remain essential concerns. Future research will focus on optimizing existing embedding algorithms, such as Least Significant Bit (LSB) modification, to minimize detectability while maximizing payload capacity. Advanced mathematical models, such as wavelet or Fourier transforms, could be integrated to embed messages more robustly across multiple layers of the media. These methods can also provide resistance to common attacks like compression and noise addition.

6.3 Neural Network-Based Parameter Approximation

The current project introduces neural networks for parameter approximation in steganographic tasks. Future work will explore the potential of generative adversarial networks (GANs) to create adaptive steganographic schemes. GANs can be trained to develop more sophisticated concealment strategies while simultaneously improving detection techniques, creating a dynamic interplay that pushes the boundaries of current methodologies.

6.4 Application to Emerging Media Formats

With the proliferation of new media formats and communication protocols, future research will extend steganographic techniques to emerging technologies, such as 3D images, holograms, and immersive virtual reality environments. These formats present unique challenges and opportunities for embedding and detecting hidden messages due to their complex structures and high data volumes.

6.5 Adherence to Cryptographic Principles

While steganography often operates independently of cryptography, future research will explore hybrid models that combine both fields. Key-dependent steganographic schemes adhering to Kerckhoffs's principle will be investigated to improve the security and robustness of hidden messages. Such models could involve cryptographic preprocessing of messages before embedding, ensuring that the concealed information remains secure even if detected.

6.6 Ethical and Legal Considerations

As steganographic techniques advance, so too does their potential for misuse. Future work will explore ethical guidelines and legal frameworks for the use of steganography in various contexts. Research will also focus on developing detection tools and forensic methods to counteract the malicious use of steganography, balancing its utility for secure communication with the need to prevent abuse.

6.7 Cross-Domain Applications

Beyond secure communication, future work will investigate steganography's application in domains like watermarking, copyright protection, and covert communication for sensitive fields like military or humanitarian operations. Research will focus on adapting existing methods to meet the unique requirements of these applications, such as durability against various forms of data degradation.

6.8 Open-Source Tool Development

To promote broader adoption and collaboration, future research will include the development of open-source libraries and tools for steganography and its detection. These tools will incorporate the algorithms, optimization techniques, and machine learning models developed in this project, serving as a resource for researchers and practitioners in the field. This project marks the beginning of an extensive research effort to explore the theoretical and practical aspects of steganography. The insights and methodologies developed here will serve as a foundation for ongoing studies, fostering innovation and addressing the evolving challenges of concealed communication in digital environments. By building upon these findings, future research will continue to expand the possibilities of steganography and its applications across diverse domains.

References

- [1] Charu C Aggarwal et al. “Neural networks and deep learning”. In: *Springer* 10 (2018), pp. 978–3.
- [2] Mohammad Ali Bani, Aman Jantan, et al. “Image encryption using block-based transformation algorithm”. In: *IJCSNS International Journal of Computer Science and Network Security* 8.4 (2008), pp. 191–197.
- [3] Christian Cachin. “An information-theoretic model for steganography”. In: *International Workshop on Information Hiding*. Springer. 1998, pp. 306–318.
- [4] Chin-Chen Chang, Ju-Yuan Hsiao, and Chi-Shiang Chan. “Finding optimal least-significant-bit substitution in image hiding by dynamic programming strategy”. In: *Pattern Recognition* 36.7 (2003), pp. 1583–1595.
- [5] Nameer N El-Emam. “Hiding a large amount of data with high security using steganography algorithm”. In: *Journal of Computer Science* 3.4 (2007), pp. 223–232.
- [6] Jessica Fridrich. *Steganography in digital media: principles, algorithms, and applications*. Cambridge University Press, 2009.
- [7] Rosziati Ibrahim and Teoh Suk Kuan. “Steganography algorithm to hide secret message inside an image”. In: *arXiv preprint arXiv:1112.2809* (2011).
- [8] Grzegorz Kłosowski and Tomasz Rymarczyk. “Using neural networks and deep learning algorithms in electrical impedance tomography”. In: *Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska* 7 (2017).
- [9] Shamim Ahmed Laskar and Kattamanchi Hemachandran. “High Capacity data hiding using LSB Steganography and Encryption”. In: *International Journal of Database Management Systems* 4.6 (2012), p. 57.
- [10] Shih-Lien Lu and Jack Kenney. “Design of most-significant-bit-first serial multiplier”. In: *Electronics Letters* 31.14 (1995), pp. 1133–1135.
- [11] Romany F Mansour and Elsaid M Abdelrahim. “An evolutionary computing enriched RS attack resilient medical image steganography model for telemedicine applications”. In: *Multidimensional Systems and Signal Processing* 30.2 (2019), pp. 791–814.
- [12] Tayana Morkel, Jan HP Eloff, and Martin S Olivier. “An overview of image steganography.” In: *ISSA*. Vol. 1. 2. 2005, pp. 1–11.
- [13] Eko Hari Rachmawanto, Christy Atika Sari, et al. “Secure image steganography algorithm based on dct with otp encryption”. In: *Journal of Applied Intelligent System* 2.1 (2017), pp. 1–11.
- [14] Phil Sallee. “Model-based steganography”. In: *International workshop on digital watermarking*. Springer. 2003, pp. 154–167.
- [15] Siddharth Singh and Tanveer J Siddiqui. “A security enhanced robust steganography algorithm for data hiding”. In: *International Journal of Computer Science Issues (IJCSI)* 9.3 (2012), p. 131.
- [16] Daniel Socek et al. “New approaches to encryption and steganography for digital videos”. In: *Multi-media Systems* 13.3 (2007), pp. 191–204.