

IMAGE DATA ENCRYPTION

DWDM Project Report

Bachelor of Technology in

Department of Applied Mathematics by

SUNAKSHI (2K17/MC/106) and VATSAL AGARWAL
(2K17/MC/119) *of branch Mathematics and Computing under*
the guidance of

Ms. Tarasha Gupta

Assistant Professor

Department of Applied Mathematics



DEPARTMENT OF APPLIED MATHEMATICS
DELHI TECHNOLOGICAL UNIVERSITY

Nov 2020

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to our guide and mentor Ms. Tarasha Gupta for her guidance at every stage of this project work. We are grateful to her for guiding us and supporting us to complete the project successfully.

We express our gratitude to the university for providing us this opportunity to explore this topic and learn several methods and facts.

Sunakshi

Vatsal Agarwal

LIST OF CONTENTS

Acknowledgment

List of Figures

- 1. OBJECTIVE**
- 2. ENCRYPTION AND DECRYPTION**
- 3. PIXEL PERMUTATION**
- 4. NEURAL NETWORK**
- 5. ARCHITECTURE**
- 6. PREPARATION LAYER/ENCRYPTION LAYER**
- 7. TRAINING NEURAL NETWORK**
- 8. SUMMARY OF ENCODER MODEL**
- 9. SUMMARY OF DECODER MODEL**
- 10.LOSS COMPUTATION**
- 11.DECRYPTION**
- 12.OPTIMIZER**
- 13.RESULTS AND CONCLUSION**
- 14.WHAT IF ORIGINAL COVER IMAGE IS AVAILABLE**

LIST OF FIGURES

Figure Number	Title
1	Encryption-Decryption
2	Pixel Permutation
3	Neural Network
4	Architecture of our project model
5	Encryption Function
6	Helpers for encryption function
7	Training Neural Network
8	Loading our dataset
9	Secret Image – Cover Image Split
10	Encoder
11	Decoder
12	Complete Neural Network
13	Loss of Reveal Layer
14	Full Loss of The Model
15	Decryption Layer Function
16	Original Image
17	Encrypted Image
18	Decrypted Image
19	Result of Neural Network
20	Errors

1. OBJECTIVE

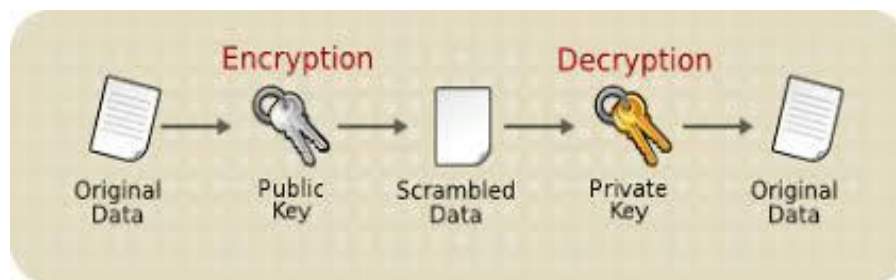
Images are widely used as part of communication since they are more effortless to process than text. The user might communicate on a network that is compromised and the security of the information becomes crucial in such cases. Hence, encryption or the scheme used to transform an image into another image that is not easily comprehensible is important to maintain the security of the information being communicated.

Our primary aim is to provide a method for image encryption using Machine Learning alongside encryption techniques. We will be using Pixel Permutation and Convolution Neural Network techniques to hide data non-uniformly and in an efficient manner.

2. ENCRYPTION AND DECRYPTION

Encryption is the process of converting normal message into meaningless message. Whereas **Decryption** is the process of converting meaningless message into its original form.

The major distinction between secret writing associated secret writing is that secret writing is that the conversion of a message into an unintelligible kind that's undecipherable unless decrypted. Whereas secret writing is that the recovery of the first message from the encrypted information.

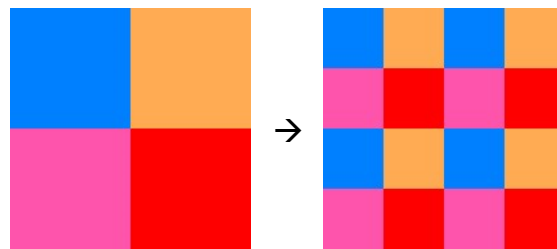


Encryption-Decryption

3. PIXEL PERMUTATION

Transposition means repositioning elements in the image. This repositioning of elements can be done at the bit, **pixel** or block level. In the bit **permutation** technique, the B bits in each **pixel** are permuted using the **permutation** key.

A permutation (rearrangement) can be described by assigning successive numbers to the objects to be permuted and then giving the order of the objects after the permutation is applied. For example, if there are eight objects 1 2 3 4 5 6 7 8 the permutation 8 7 6 5 4 3 2 1 reverses the order of the objects.

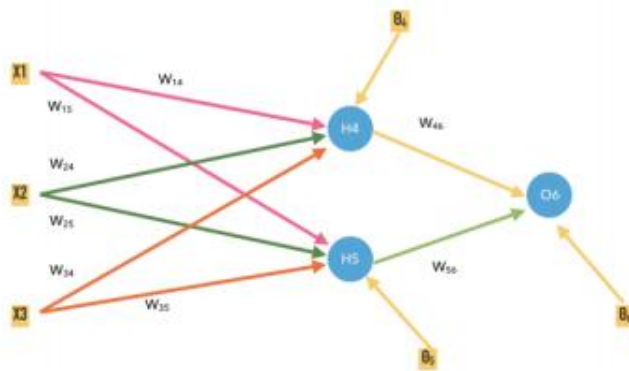


Pixel Permutation

4. NEURAL NETWORK

The Artificial Neural Network resembles a human brain in its structure and functioning. The decision-making algorithm that an artificial neural network uses is holistic, which depends on the aggregate of the entire input patterns and is also motivated by the human thinking process. ANN is greatly used in fields of pattern recognition, classification, etc. ANN gives quite accurate results provided they are trained well. The neural systems define their potential because of their hugely lateral structure, and the capacity to use the experience to learn. ANN gives quite accurate results provided they are trained well. The exactness of the results relies upon the efficiency of training, which thus relies on the meticulousness and extent of the

training. The information picked up by the learning experience is put away as association weights, which are utilized to provide output for test data.



Neural Network

Above figure, depicts a simple neural network with a number of layers - 3 namely - an input layer at the leftmost side, an output layer at the rightmost side, and a hidden layer in between. When the count of hidden layers is increased it constructs a more advanced and complex neural network. To a certain extent, increasing the count of hidden layers helps in finding a better decision boundary and accuracy of the network. The structure may have a few 'layers' of neurons and the overall structure may either be feedback or feedforward structure. If we just have to classify into linearly separable classes, a solitary layer perceptron classifier is quite sufficient. In any case, if the classification is more complex a multi-layer perceptron that acts as a multi-layer feed-forward system is generally accepted to get better accuracy. In a multi-layer perceptron, the association weights are randomly allocated at the outset and continuously adjusted to minimize the overall mean square mistake in the framework. The weight starts updating from the last layer and advances towards the opposite side until it reaches the first layer. This process of updating the weight is termed as the backpropagation learning algorithm (BPA), which is a supervised learning method. The chief objective of weight update is to maximize the degree of reduction of error, and subsequently, it is named as 'gradient descent' algorithm. The weight updating is done in 'small' steps; the step size is picked heuristically, as there is no precise guideline for its choice. Artificial Neural Network has certain advantages over other machine learning techniques. The biggest one is that it learns by itself. It can complete tasks that a linear program will fail to do or complete with lesser efficiency. Another major benefit is that even if a component of the neural network fails to execute, even then the neural network will produce efficient results due to parallel execution. Also, once the model is established it will discover its correct values on its own and does not need to be remodelled. Once shaped, the

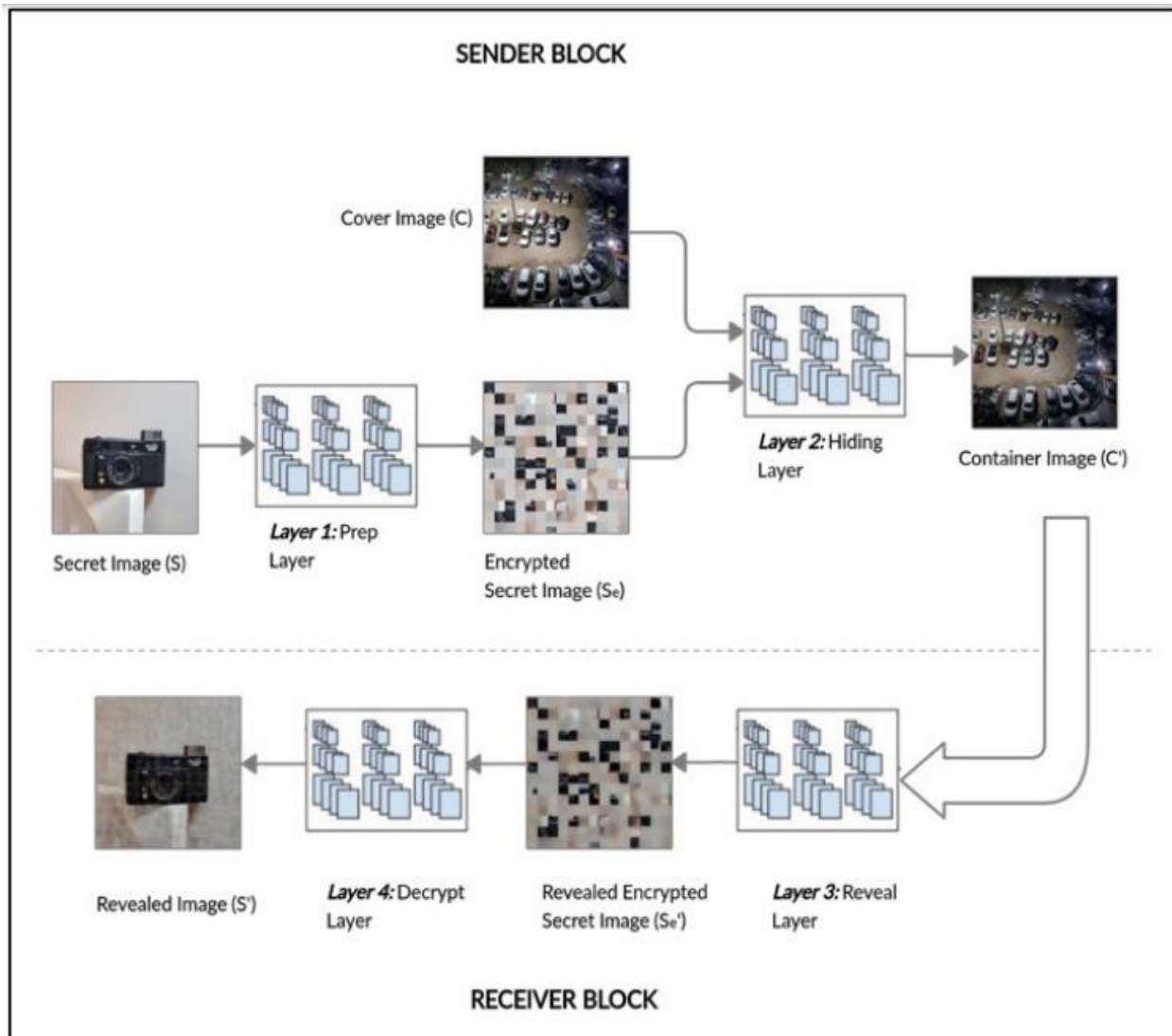
prediction time of the neural network is too small. Any number of inputs and layers can be trained using neural networks and it works fine for a huge number of tuples as well. ANN is adaptable and can be used for regression problems as well. ANN can be used to determine any problem involving analytical attributes. It is stable in problems having a large number of features. A big drawback in ANN is that it behaves like a black box and hence we don't have any report concerning what is going inside the network. We don't know what impact each independent attribute is having on the dependent attribute. Another factor is the cost required in implementing it on a conventional CPU. It is expensive both in terms of time required to train and mathematically compute as well as in terms of the capital required to have a CPU with requisite highlights. Another big concern related to ANN is that it relies heavily on training data, so there may be an issue of overfitting and generalization. Although quite efficient, if we have a large number of target attributes a complex neural network will be established which may require high processing time.

5. ARCHITECTURE

Our complete model for this project is made up of four layers:

1. Prep Layer
2. Hiding Layer
3. Reveal Layer
4. Decrypt Layer

The first layer, Prep Layer, prepares the secret image to be hidden. This layer serves various purposes. Firstly, if the secret image is smaller than the cover image, it increases the size of the secret image to the size of the cover image, hence distributing the secret image's bits across the entire pixels. The most important purpose of this layer was to finally embed the secret image to the encrypted image so as to avoid any leak of the secret message.



Architecture of our project model

The second layer, Hiding Layer, takes the output of the Prep Layer as input and a cover image so as to produce the Container Image. The input to this network is a square pixel image, with depth considered of RGB channels of the cover image and the reconstructed channels of the secret image. These two layers together form the Sender Block. The Container image produced can be shared with the receiver. The third Layer, Reveal Layer, is used by the receiver to produce back the encrypted image. This layer takes in the Container image as input and removes the cover image to generate the encrypted secret image. The fourth Layer, Decrypt Layer, takes in the output of the Reveal Layer and decrypt the image to finally reveal the secret image. The third and fourth layer together forms the Receiver Block.

6. PREPARATION LAYER/ENCRYPTION LAYER

This layer marks the beginning of our project. The basic aim of this layer is to take the secret image as input and cut down it into small blocks and scramble it. These cut down parts of images are referred to as tiles of square shape. We started by performing a series of experiments in which take some images and cut down them into various number of tiles - [4,9,16,64,196]. We first generated a set of numbers from 1 to number of tiles chosen and then shuffled them. We stored this info and then cut the tiles into specified number of tiles. We then generate a new image using tiles index from the shuffled dataset and the original tiles. This give us a complex image which is difficult to comprehend and analyze. Thus, it enhances the security of our network.

```
In [14]: def cuts_to_encrypted(cuts,tiles,original_size,name,order):
        k=0
        blank_image = Image.new('RGB',original_size)
        for i in range(0,int(sqrt(cuts))):
            for j in range(0,int(sqrt(cuts))):
                tile_index = order[k]
                im = tiles[tile_index].image
                blank_image.paste(im,(j*int(original_size[0]/sqrt(cuts)),i*int(original_size[1]/sqrt(cuts))))
                k = k+1

        string = name+"-encrypted-"+str(cuts)+".jpg"
        blank_image.save(string)
```

Encryption Function

```
In [16]: def dataset_generator(name):
        l = [4,9,16,64,196]
        for j in l:
            tiles = image_slicer.slice('./OSI/'+str(name)+'.jpg',j,save=False)
            original = Image.open('./OSI/'+str(name)+'.jpg')
            print(j)
            order = orders[j]
            cuts_to_encrypted(j,tiles,original.size,name,order)
```

```
In [17]: def dataset_gen_helper():
        for i in range(1,8):
            #print(cuts)
            dataset_generator(str(i))
```

```
In [18]: def order_gen():
        l = [4,9,16,64,196]
        orders = {}
        for j in l:
            order = [x for x in range(j)]
            shuffle(order)
            orders[j] = order

        return orders
```

```
In [19]: orders = order_gen()
```

```
In [20]: dataset_gen_helper()
```

Helpers for encryption function

After performing our experiment on the number of tiles in which image should be broken, we find out that 196 number of tiles from a image is sufficient enough to make it incomprehensible. We thus performed our further classification on the basis of these numbers.

7. TRAINING NEURAL NETWORK

Our process started by loading our dataset that is Flickr32. Our dataset contains total of 31,783 images having the standard RGB Scale. The dataset is publicly available at [website](#). Each image is further divided into blocks which are later scrambled during our first layer of preparation. We will be loading our dataset into two lists X_train and X_test and later we will convert them both into numpy arrays for easier calculations ahead.

```
In [3]: def load_dataset_small(num_images_per_class_train=30000, num_images_test=50):
        """Loads training and test datasets, from Tiny ImageNet Visual Recognition Challenge.

        Arguments:
            num_images_per_class_train: number of images per class to load into training dataset.
            num_images_test: total number of images to load into training dataset.
        """
        X_train = []
        X_test = []

        # Create training set.
        for c in os.listdir(TRAIN_DIR):
            print(c)
            if c == '.DS_Store':
                continue
            c_dir = os.path.join(TRAIN_DIR, c)
            c_imgs = os.listdir(c_dir)
            random.shuffle(c_imgs)
            for img_name_i in c_imgs[0:num_images_per_class_train]:
                img_i = image.load_img(os.path.join(c_dir, img_name_i))
                x = image.img_to_array(img_i)
                X_train.append(x)
            random.shuffle(X_train)

        # Create test set.
        test_dir = os.path.join(TEST_DIR, 'images')
        test_imgs = os.listdir(test_dir)
        random.shuffle(test_imgs)
        for img_name_i in test_imgs[0:num_images_test]:
            img_i = image.load_img(os.path.join(test_dir, img_name_i))
            x = image.img_to_array(img_i)
            X_test.append(x)

        # Return train and test data as numpy arrays.
        return np.array(X_train), np.array(X_test)
```

Training Neural Network

The shape of our train dataset and test dataset is:

```
In [4]: # Load dataset.
X_train_orig, X_test_orig = load_dataset_small()

# Normalize image vectors.
X_train = X_train_orig/255.
X_test = X_test_orig/255.

# Print statistics.
print ("Number of training examples = " + str(X_train.shape[0]))
print ("Number of test examples = " + str(X_test.shape[0]))
print ("X_train shape: " + str(X_train.shape)) # Should be (train_size, 64, 64, 3).

images
.DS_Store
Number of training examples = 50
Number of test examples = 50
X_train shape: (50, 256, 256, 3)
```

Loading our dataset

We split our dataset into two parts. Two equal halves are made where one set is used as Secret Image while the other half is used as cover image.

```
In [5]: # We split training set into two halves.
# First half is used for training as secret images, second half for cover images.

# S: secret image
input_S = X_test[0:X_test.shape[0] // 2]

# C: cover image
input_C = X_test[X_test.shape[0] // 2:]
```

Secret Image – Cover Image Split

In the next step we start building our models. We build models for three layers: Preparation Network, Hiding Layer and Reveal Layer. The goal of these layers is to encode the image data of secret image inside the cover image, and thus generating a container image, the container image should closely resemble the cover image and while this all process goes on, we should still be able to get back our secret image. The revealed secret image should be as close as possible to our secret image. Our first layer or the Preparation Network prepares our data from the secret image. This data needs to be overlapped with the cover image and then passed as input to the hiding layer. The hiding layer converts the secret image into a concatenated image called the container image, and at last this concatenated image is passed as input to the reveal layer which decodes it back to the secret image.

Preparation layer and the hiding layer are grouped together to form the ‘Encoder’ model. Finally, this encoder model is provided with 2 inputs, ‘input_S’ and ‘input_C’, and this encoder model gives out an output called ‘output_Cprime’ which is our container image.

```
def make_encoder(input_size):
    input_S = Input(shape=(input_size))
    input_C = Input(shape=(input_size))

    # Preparation Network
    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_prep0_3x3')(input_S)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_prep0_4x4')(input_S)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_prep0_5x5')(input_S)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_prep1_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_prep1_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_prep1_5x5')(x)
    x = concatenate([x3, x4, x5])

    x = concatenate([input_C, x])

    # Hiding network
    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_hid0_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_hid0_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_hid0_5x5')(x)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_hid1_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_hid1_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_hid1_5x5')(x)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_hid2_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_hid2_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_hid2_5x5')(x)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_hid3_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_hid3_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_hid3_5x5')(x)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_hid4_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_hid4_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_hid5_5x5')(x)
    x = concatenate([x3, x4, x5])

    output_Cprime = Conv2D(3, (3, 3), strides = (1, 1), padding='same', activation='relu', name='output_C')(x)

    return Model(inputs=[input_S, input_C],
                  outputs=output_Cprime,
                  name = 'Encoder')
```

Encoder

Now the reveal layer forms the ‘Decoder’ model.

```
def make_decoder(input_size, fixed=False):

    # Reveal network
    reveal_input = Input(shape=(input_size))

    # Adding Gaussian noise with 0.01 standard deviation.
    input_with_noise = GaussianNoise(0.01, name='output_C_noise')(reveal_input)

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_rev0_3x3')(input_with_noise)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_rev0_4x4')(input_with_noise)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_rev0_5x5')(input_with_noise)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_rev1_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_rev1_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_rev1_5x5')(x)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_rev2_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_rev2_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_rev2_5x5')(x)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_rev3_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_rev3_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_rev3_5x5')(x)
    x = concatenate([x3, x4, x5])

    x3 = Conv2D(50, (3, 3), strides = (1, 1), padding='same', activation='relu', name='conv_rev4_3x3')(x)
    x4 = Conv2D(10, (4, 4), strides = (1, 1), padding='same', activation='relu', name='conv_rev4_4x4')(x)
    x5 = Conv2D(5, (5, 5), strides = (1, 1), padding='same', activation='relu', name='conv_rev5_5x5')(x)
    x = concatenate([x3, x4, x5])

    output_Sprime = Conv2D(3, (3, 3), strides = (1, 1), padding='same', activation='relu', name='output_S')(x)
```

Decoder

Finally, the structure of our complete model is:

```
# full model.
def make_model(input_size):
    input_S = Input(shape=(input_size))
    input_C = Input(shape=(input_size))

    encoder = make_encoder(input_size)

    decoder = make_decoder(input_size)
    decoder.compile(optimizer='adam', loss=rev_loss)
    decoder.trainable = False

    output_Cprime = encoder([input_S, input_C])
    output_Sprime = decoder(output_Cprime)

    autoencoder = Model(inputs=[input_S, input_C],
                        outputs=concatenate([output_Sprime, output_Cprime]))
    autoencoder.compile(optimizer='adam', loss=full_loss)

    return encoder, decoder, autoencoder
```

```
In [8]: encoder_model, reveal_model, autoencoder_model = make_model(input_S.shape[1:])
```

Complete Neural Network

8. SUMMARY OF THE ENCODER MODEL

Applications ▾ Firefox Web Browser ▾ Nov 26 03:42

DWDM/deep-steg-samp x New Tab x NeuralNet - Jupyter No: x +

localhost:8888/notebooks/DWDM/deep-steg-sample/NeuralNet.ipynb

Jupyter NeuralNet Last Checkpoint: 8 minutes ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

In [17]: encoder_model.summary()

Model: "Encoder"

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	[(None, 256, 256, 3) 0]		
conv_prep0_3x3 (Conv2D)	(None, 256, 256, 50) 1400		input_3[0][0]
conv_prep0_4x4 (Conv2D)	(None, 256, 256, 10) 490		input_3[0][0]
conv_prep0_5x5 (Conv2D)	(None, 256, 256, 5) 380		input_3[0][0]
concatenate (Concatenate)	(None, 256, 256, 65) 0		conv_prep0_3x3[0][0] conv_prep0_4x4[0][0] conv_prep0_5x5[0][0]
conv_prep1_3x3 (Conv2D)	(None, 256, 256, 50) 29300		concatenate[0][0]
conv_prep1_4x4 (Conv2D)	(None, 256, 256, 10) 10410		concatenate[0][0]
conv_prep1_5x5 (Conv2D)	(None, 256, 256, 5) 8130		concatenate[0][0]
input_4 (InputLayer)	[(None, 256, 256, 3) 0]		
concatenate_1 (Concatenate)	(None, 256, 256, 65) 0		conv_prep1_3x3[0][0] conv_prep1_4x4[0][0] conv_prep1_5x5[0][0]
concatenate_2 (Concatenate)	(None, 256, 256, 68) 0		input_4[0][0] concatenate_1[0][0]
conv_hid0_3x3 (Conv2D)	(None, 256, 256, 50) 30650		concatenate_2[0][0]
conv_hid0_4x4 (Conv2D)	(None, 256, 256, 10) 10890		concatenate_2[0][0]

Applications

Firefox Web Browser

Nov 26 03:42

DWDM/deep-steg-samp

New Tab

NeuralNet - Jupyter Not

+

localhost:8888/notebooks/DWDM/deep-steg-sample/NeuralNet.ipynb

jupyter

NeuralNet

Last Checkpoint: 8 minutes ago (autosaved)

Logout

File

Edit

View

Insert

Cell

Kernel

Widgets

Help

Trusted

Python 3

Run

Code

In [17]:

encoder_model.summary()

Model: "Encoder"

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 256, 256, 3) 0		
conv_prep0_3x3 (Conv2D)	(None, 256, 256, 50) 1400		input_3[0][0]
conv_prep0_4x4 (Conv2D)	(None, 256, 256, 10) 490		input_3[0][0]
conv_prep0_5x5 (Conv2D)	(None, 256, 256, 5) 380		input_3[0][0]
concatenate (Concatenate)	(None, 256, 256, 65) 0		conv_prep0_3x3[0][0] conv_prep0_4x4[0][0] conv_prep0_5x5[0][0]
conv_prep1_3x3 (Conv2D)	(None, 256, 256, 50) 29300		concatenate[0][0]
conv_prep1_4x4 (Conv2D)	(None, 256, 256, 10) 10410		concatenate[0][0]
conv_prep1_5x5 (Conv2D)	(None, 256, 256, 5) 8130		concatenate[0][0]
input_4 (InputLayer)	(None, 256, 256, 3) 0		
concatenate_1 (Concatenate)	(None, 256, 256, 65) 0		conv_prep1_3x3[0][0] conv_prep1_4x4[0][0] conv_prep1_5x5[0][0]
concatenate_2 (Concatenate)	(None, 256, 256, 68) 0		input_4[0][0] concatenate_1[0][0]
conv_hid0_3x3 (Conv2D)	(None, 256, 256, 50) 30650		concatenate_2[0][0]
conv_hid0_4x4 (Conv2D)	(None, 256, 256, 10) 10890		concatenate_2[0][0]

Applications

Firefox Web Browser

Nov 26 03:43

DWDM/deep-steg-samp

New Tab

NeuralNet - Jupyter Not

+

localhost:8888/notebooks/DWDM/deep-steg-sample/NeuralNet.ipynb

jupyter

NeuralNet

Last Checkpoint: 10 minutes ago (autosaved)

Logout

File

Edit

View

Insert

Cell

Kernel

Widgets

Help

Trusted

Python 3

Run

Code

conv_hid3_3x3 (Conv2D)

(None, 256, 256, 50) 29300

concatenate_3[0][0]

conv_hid3_4x4 (Conv2D)

(None, 256, 256, 10) 10410

concatenate_5[0][0]

conv_hid3_5x5 (Conv2D)

(None, 256, 256, 5) 8130

concatenate_5[0][0]

concatenate_6 (Concatenate)

(None, 256, 256, 65) 0

conv_hid3_3x3[0][0]
conv_hid3_4x4[0][0]
conv_hid3_5x5[0][0]

conv_hid4_3x3 (Conv2D)

(None, 256, 256, 50) 29300

concatenate_6[0][0]

conv_hid4_4x4 (Conv2D)

(None, 256, 256, 10) 10410

concatenate_6[0][0]

conv_hid5_5x5 (Conv2D)

(None, 256, 256, 5) 8130

concatenate_6[0][0]

concatenate_7 (Concatenate)

(None, 256, 256, 65) 0

conv_hid4_3x3[0][0]
conv_hid4_4x4[0][0]
conv_hid5_5x5[0][0]

output_C (Conv2D)

(None, 256, 256, 3) 1758

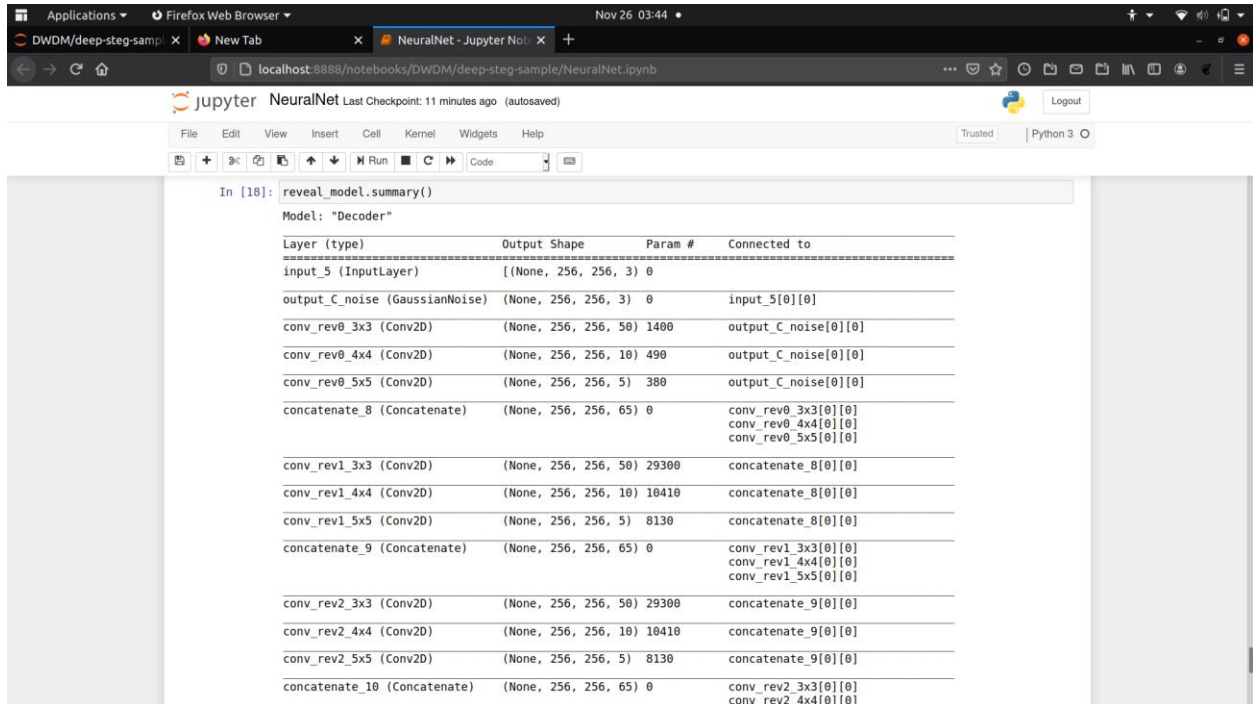
concatenate_7[0][0]

Total params: 293,273

Trainable params: 293,273

Non-trainable params: 0

9. SUMMARY OF THE DECODER MODEL

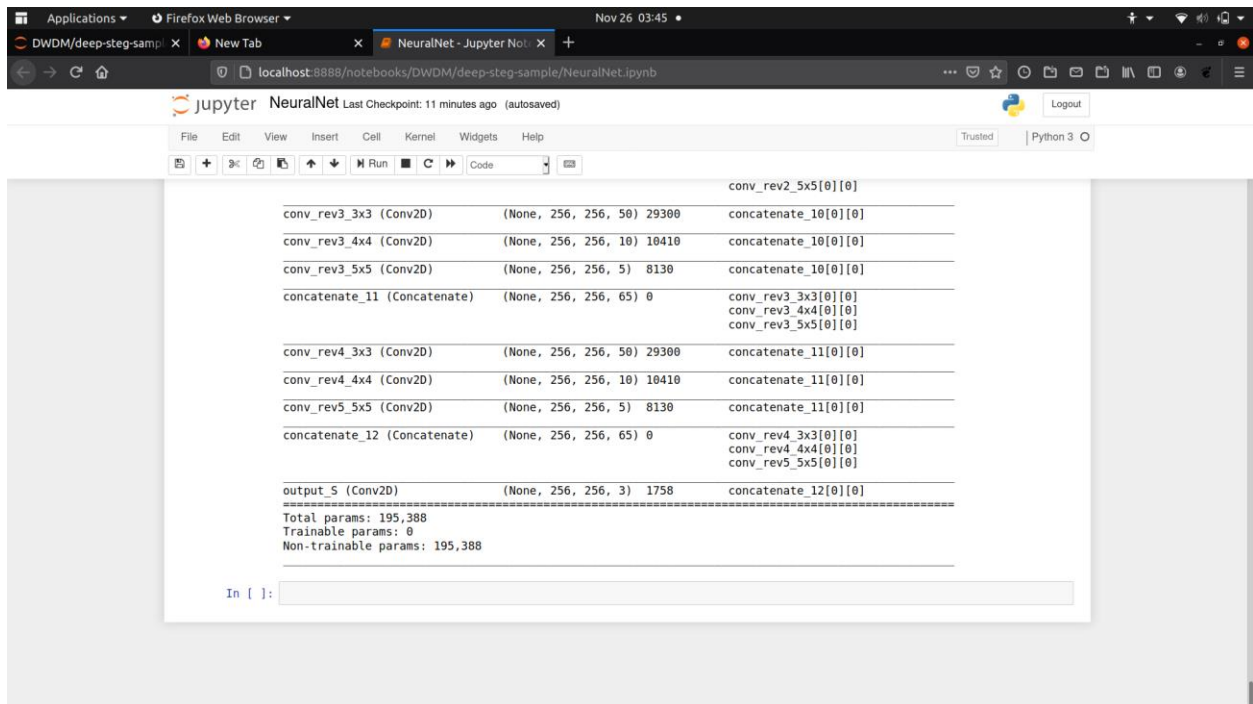


The screenshot shows a Jupyter Notebook interface with a Firefox browser window. The notebook is titled 'NeuralNet' and shows the output of the `reveal_model.summary()` command. The output is a detailed summary of the 'Decoder' model architecture, including layer types, output shapes, parameter counts, and connections.

```
In [18]: reveal_model.summary()

Model: "Decoder"

Layer (type)                 Output Shape          Param #   Connected to
-----
input_5 (InputLayer)         [(None, 256, 256, 3)] 0
output_C_noise (GaussianNoise) (None, 256, 256, 3) 0 input_5[0][0]
conv_rev0_3x3 (Conv2D)        (None, 256, 256, 50) 1400 output_C_noise[0][0]
conv_rev0_4x4 (Conv2D)        (None, 256, 256, 10) 490 output_C_noise[0][0]
conv_rev0_5x5 (Conv2D)        (None, 256, 256, 5) 380 output_C_noise[0][0]
concatenate_8 (Concatenate)   (None, 256, 256, 65) 0 conv_rev0_3x3[0][0]
conv_rev0_4x4[0][0]
conv_rev0_5x5[0][0]
conv_rev1_3x3 (Conv2D)        (None, 256, 256, 50) 29300 concatenate_8[0][0]
conv_rev1_4x4 (Conv2D)        (None, 256, 256, 10) 10410 concatenate_8[0][0]
conv_rev1_5x5 (Conv2D)        (None, 256, 256, 5) 8130 concatenate_8[0][0]
concatenate_9 (Concatenate)   (None, 256, 256, 65) 0 conv_rev1_3x3[0][0]
conv_rev1_4x4[0][0]
conv_rev1_5x5[0][0]
conv_rev2_3x3 (Conv2D)        (None, 256, 256, 50) 29300 concatenate_9[0][0]
conv_rev2_4x4 (Conv2D)        (None, 256, 256, 10) 10410 concatenate_9[0][0]
conv_rev2_5x5 (Conv2D)        (None, 256, 256, 5) 8130 concatenate_9[0][0]
concatenate_10 (Concatenate)  (None, 256, 256, 65) 0 conv_rev2_3x3[0][0]
conv_rev2_4x4[0][0]
```



The screenshot shows the continuation of the model summary from the previous image. It lists the remaining layers of the decoder, including convolutional and concatenation layers, and provides the total number of parameters and trainable/non-trainable counts.

```
conv_rev2_5x5[0][0]
conv_rev3_3x3 (Conv2D)        (None, 256, 256, 50) 29300 concatenate_10[0][0]
conv_rev3_4x4 (Conv2D)        (None, 256, 256, 10) 10410 concatenate_10[0][0]
conv_rev3_5x5 (Conv2D)        (None, 256, 256, 5) 8130 concatenate_10[0][0]
concatenate_11 (Concatenate)  (None, 256, 256, 65) 0 conv_rev3_3x3[0][0]
conv_rev3_4x4[0][0]
conv_rev3_5x5[0][0]
conv_rev4_3x3 (Conv2D)        (None, 256, 256, 50) 29300 concatenate_11[0][0]
conv_rev4_4x4 (Conv2D)        (None, 256, 256, 10) 10410 concatenate_11[0][0]
conv_rev5_5x5 (Conv2D)        (None, 256, 256, 5) 8130 concatenate_11[0][0]
concatenate_12 (Concatenate)  (None, 256, 256, 65) 0 conv_rev4_3x3[0][0]
conv_rev4_4x4[0][0]
conv_rev5_5x5[0][0]
output_5 (Conv2D)            (None, 256, 256, 3) 1758 concatenate_12[0][0]
=====
Total params: 195,388
Trainable params: 0
Non-trainable params: 195,388

In [ ]:
```

10. LOSS COMPUTATION

In our model, we will be discussing two types of losses – the first one that is present only in the reveal layer and the other one that is present in the complete model. Reveal layer loss is used to determine how much does the revealed secret image differs from the original secret image. Basically, it is the difference in the revealed secret image and initial secret image.

```
# Loss for reveal network
def rev_loss(s_true, s_pred):
    # Loss for reveal network is: beta * |S-S'|
    return beta * K.sum(K.square(s_true - s_pred))
```

Loss of Reveal Layer

The other loss is the full loss that is computer over the whole model and it basically contains both the parts – first that belongs to difference in the revealed secret image and the original secret image and other that belongs to difference in the container image and the cover image. This is calculated because there should be minimum difference between the container image and the cover image otherwise, an intruder can identify with the naked eyes the difference between the two images and check out there is some secret message stored here.

```
# Loss for the full model, used for preparation and hiding networks
def full_loss(y_true, y_pred):
    # Loss for the full model is: |C-C'| + beta * |S-S'|
    s_true, c_true = y_true[...,0:3], y_true[...,3:6]
    s_pred, c_pred = y_pred[...,0:3], y_pred[...,3:6]

    s_loss = rev_loss(s_true, s_pred)
    c_loss = K.sum(K.square(c_true - c_pred))

    return s_loss + c_loss
```

Full Loss of The Model

The value of ‘Beta’ here is taken as 1.0 because we want to place the weight of cover image loss and the secret image loss as equal.

11. DECRYPTION LAYER

Decryption layer is used to reverse the task that is done by encryption layer. It is used to make the secret image again comprehensible and readable. For this the receiver should know in advance the shuffled set of tiles index, so that he can perform the back operations and again assemble the secret image back. Here again we first need to break our shuffled image into 196 tiles. Then we make a new image of same size and using the shuffled set of indexes and scrambled image we make a new image which is same as the original image.

```
In [15]: def cuts_to_original(cuts,tiles,original_size,name,order):
        k=0
        blank_image = Image.new('RGB',original_size)
        for i in range(0,int(sqrt(cuts))):
            for j in range(0,int(sqrt(cuts))):
                print("k :"+str(k))
                tile_index = order.index(k)
                print('tile index : '+str(tile_index))
                im = tiles[tile_index].image
                blank_image.paste(im,(j*int(original_size[0]/sqrt(cuts)),i*int(original_size[1]/sqrt(cuts))))
                k = k+1

        blank_image.save(str(name)+'-decrypted.jpg')
```

Decryption Layer Function

12. OPTIMIZER

The optimizer in a model has an objective to adjust the weights for every edge of network to reduce the value of the loss function to an optimal value. Its motive is to find global minima for the loss function.

There are different optimizers for different usage for instance – Momentum, Adagrad, Nesterov Accelerated Gradient, Adam, Adadelata, and so forth. The proposed model operates on ADAM – Adaptive Moment Estimation optimizer which computes a different rate of learning for every parameter. The main advantage of ADAM is that it doesn't need heavy computational power and has minimal memory necessities. The explanation for not bringing Momentum, Nestorov analyzers and Stochastic Gradient Descent into action are on the grounds that the dataset utilized is meager. Other versatile learning techniques, for example, RMSprop, Adadelata, and Adagrad were beaten by the ADAM analyzer settling on it as the perfect decision.

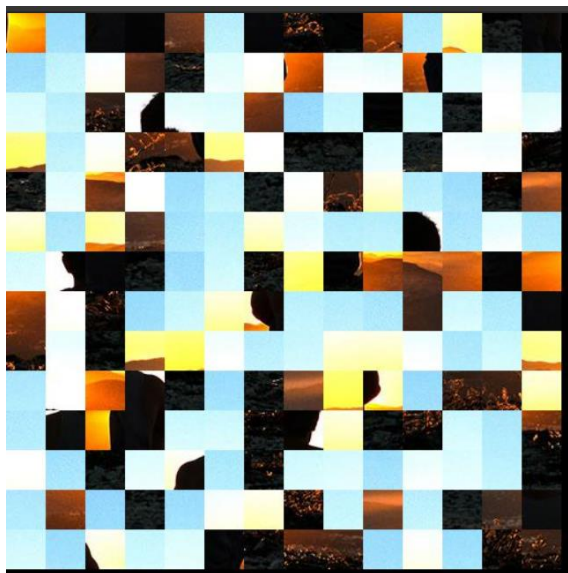
13. RESULTS AND CONCLUSION

ENCRYPTION/DECRYPTION

Results of encryption and decryption layer can be shown by showing three images. First that is our original image, second the encrypted image that came out after computation and third is again decrypted image. If first and the final image are same that means that the decryption layer worked well and since we are breaking an image into 196 tiles and then shuffling them, this leads to $196!$ possible combinations. Based on the fact that a computer can perform 10^{16} calculations per second, it will take around 10342 years to get to the exact results. This fact also ensures the fact that using pixel permutation as our first layer of encryption is quite efficient.



Original Image



Encrypted Image



Decrypted Image

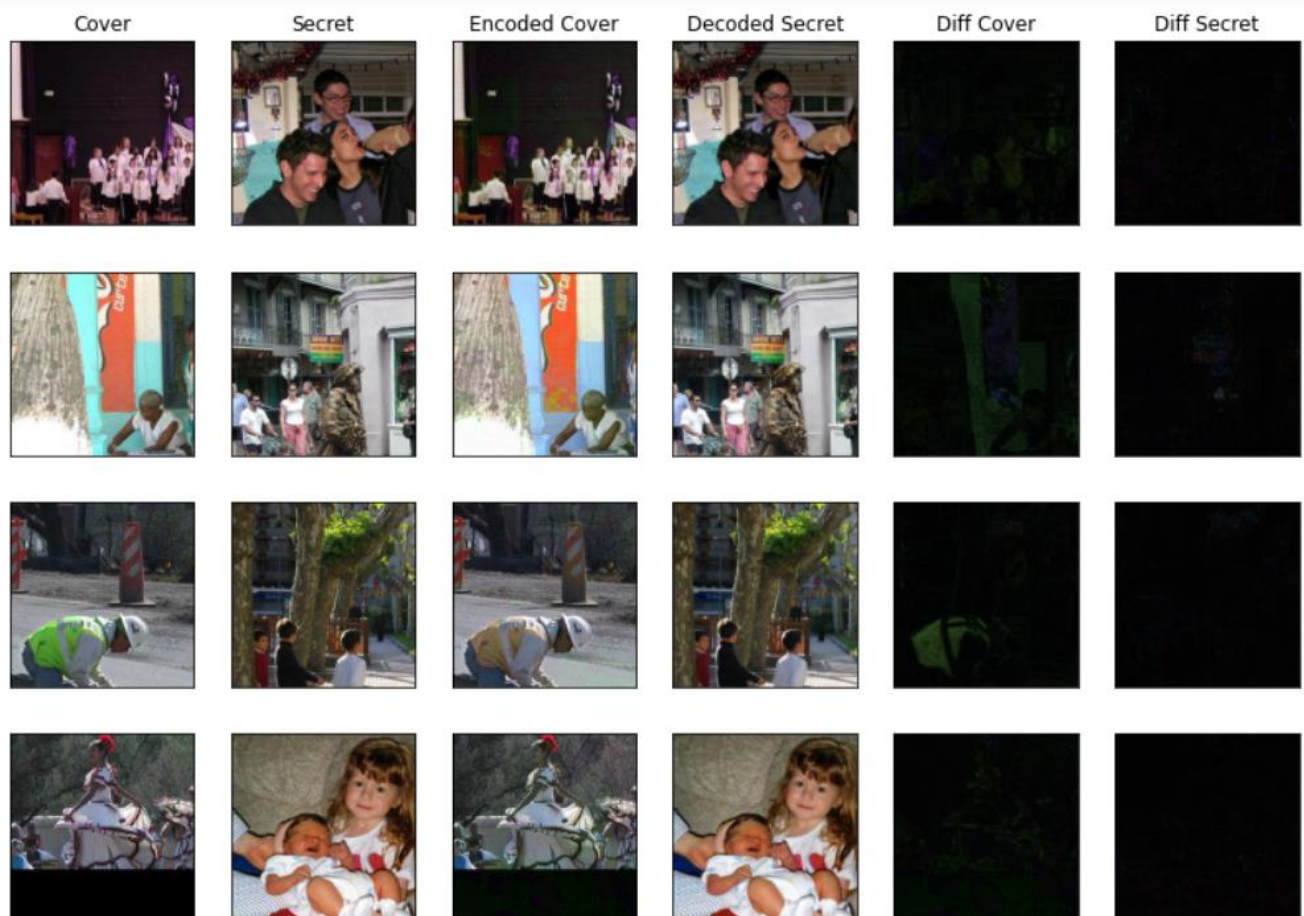
```

In [40]: # Configs for results display
# Show images in gray scale
SHOW_GRAY = False
# Show difference between predictions and ground truth.
SHOW_DIFF = True
# Diff enhance magnitude
ENHANCE = 1
# Number of secret and cover pairs to show.
n = 6

def rgb2gray(rgb):
    return np.dot(rgb[...,:3], [0.299, 0.587, 0.114])
def show_image(img, n_rows, n_col, idx, gray=False, first_row=False, title=None):
    ax = plt.subplot(n_rows, n_col, idx)
    if gray:
        plt.imshow(rgb2gray(img), cmap = plt.get_cmap('gray'))
    else:
        plt.imshow(img)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    if first_row:
        plt.title(title)
plt.figure(figsize=(14, 15))
rand_idx = [random.randint(0, len(input_S)) for x in range(n)]
# for i, idx in enumerate(range(0, n)):
for i, idx in enumerate(rand_idx):
    n_col = 6 if SHOW_DIFF else 4
    show_image(input_C[idx], n, n_col, i * n_col + 1, gray=SHOW_GRAY, first_row=i==0, title='Cover')
    show_image(input_S[idx], n, n_col, i * n_col + 2, gray=SHOW_GRAY, first_row=i==0, title='Secret')
    show_image(decoded_C[idx], n, n_col, i * n_col + 3, gray=SHOW_GRAY, first_row=i==0, title='Encoded Cover')
    show_image(decoded_S[idx], n, n_col, i * n_col + 4, gray=SHOW_GRAY, first_row=i==0, title='Decoded Secret')

    if SHOW_DIFF:
        show_image(np.multiply(diff_C[idx], ENHANCE), n, n_col, i * n_col + 5, gray=SHOW_GRAY, first_row=i==0, title='Diff Cover')
        show_image(np.multiply(diff_S[idx], ENHANCE), n, n_col, i * n_col + 6, gray=SHOW_GRAY, first_row=i==0, title='Diff Secret')
plt.show()

```

Result of Neural Network

```
In [38]: # Print pixel-wise average errors in a 256 scale.
S_error, C_error = pixel_errors(input_S, input_C, decoded_S, decoded_C)

print ("S error per pixel [0, 255]:", S_error)
print ("C error per pixel [0, 255]:", C_error)
```

```
S error per pixel [0, 255]: 5.3680058
C error per pixel [0, 255]: 9.501964
```

Errors

14. WHAT IF ORIGINAL COVER IMAGE IS AVAILABLE?

Basically, without the first layer of encryption, data can be revealed by taking the difference between the original cover image and the reconstructed image. It won't be visible directly, but after enhancing the image and converting it into grayscale, the shade of actual image may be revealed.

But if we use the first layer of encryption, the secret is impossible to obtain as after enhancing and converting the residual image into grayscale we will see only scrambled image which is difficult to comprehend even in initial stage.