Implementation

Functions

- **config.py -** It contains the initialization of the filters, key length, message length, batch size, epoch and the learning rate. These variables will be given as input in the training.
- **layers.py** - It is used for the initialization of the structure of convolution layers of A,B and E. It imports the filter values from config.py.
- **utils.py** - It contains the functions to generate n random messages and keys and and also the function for the Xavier Glotrot initialization of weights of the neural network layers.
- **model.py** - It contains the class used for building the model with the layers and the weights using the functions imported from layers.py, config.py, utils.py. It also contains the training functions implementing the loss and the optimizer functions. Additionally it also has the function (predict()) used for for testing data.
- **test.txt -** It is the testing text file .
- **main.ipynb -** It acts as an entry point for the code and contains the instances of the class present in the model.py. It also contains functions used for convertion of text to bits and bits to text.

Flow of the code

- We are first intializing batch size as 512, message and key length as 8 in the utils.py and also main.ipynb.
- We make use of tensorflow sessions so all the variables and functions are not valid outside of this session.
- We create a **crypto_net** [instance of **CryptoNet** class in **model.py**] and initilaize the values.

```
crypto_net = CryptoNet(sess, msg_len=MSG_LEN, epochs=NUM_EPOCHS,
                       batch_size=BATCH_SIZE, learning_rate=LEARNING_RATE)

crypto_net.train()
```

- We then train it using **crypto_net.train()**.

```
 def train(self):
        # Loss Functions
        self.decrypt_err_eve = tf.reduce_mean(tf.abs(self.msg -
self.eve_output))
        self.decrypt_err_bob = tf.reduce_mean(tf.abs(self.msg -
self.bob_output))
        self.loss_bob = self.decrypt_err_bob + (1. - self.decrypt_err_eve) ** 2.

        # Get training variables corresponding to each network
        self.t_vars = tf.trainable_variables()
        self.alice_or_bob_vars = [var for var in self.t_vars if 'alice_' in
var.name or 'bob_' in var.name]
        self.eve_vars = [var for var in self.t_vars if 'eve_' in var.name]

        # Build the optimizers
        self.bob_optimizer =
tf.train.AdamOptimizer(self.learning_rate).minimize(
            self.loss_bob, var_list=self.alice_or_bob_vars)
        self.eve_optimizer =
tf.train.AdamOptimizer(self.learning_rate).minimize(
            self.decrypt_err_eve, var_list=self.eve_vars)

        self.bob_errors, self.eve_errors = [], []

        # Begin Training
        tf.global_variables_initializer().run()
        for i in range(self.epochs):
            iterations = 30

            print('Training Alice and Bob, Epoch:', i + 1)
            bob_loss, _ = self._train('bob', iterations)
            self.bob_errors.append(bob_loss)

            print('Training Eve, Epoch:', i + 1)
            _, eve_loss = self._train('eve', iterations)
            self.eve_errors.append(eve_loss)

        self.plot_errors()
```

- Each network first trains to calculate B’s error maintaining E’s error as constant and the trains to calculates E’s error maintaing B’s error as constant by using **self._train()**
- This processes continues for all the iterations and epochs.

```
def _train(self, network, iterations):
        bob_decrypt_error, eve_decrypt_error = 1., 1.

        bs = self.batch_size
        # Train Eve for two minibatches to give it a slight computational edge
        if network == 'eve':
            bs *= 2

        for i in range(iterations):
            msg_in_val, key_val = gen_data(n=bs, msg_len=self.msg_len,
                                                    key_len=self.key_len)

            if network == 'bob':
                _, decrypt_err = self.sess.run([self.bob_optimizer,
                                  self.decrypt_err_bob],
                                               feed_dict={self.msg: msg_in_val,
                                                        self.key: key_val})
#                 bob_decrypt_error =  decrypt_err
#                 print(bob_decrypt_error,'b')
                bob_decrypt_error = min(bob_decrypt_error, decrypt_err)


            elif network == 'eve':

                _, decrypt_err = self.sess.run([self.eve_optimizer,
                                   self.decrypt_err_eve],
                                               feed_dict={self.msg: msg_in_val,
                                                     self.key: key_val})

#                 eve_decrypt_error =  decrypt_err
#                 print(eve_decrypt_error,'e')

                eve_decrypt_error = min(eve_decrypt_error, decrypt_err)


        print(bob_decrypt_error,'b')
        print(eve_decrypt_error,'e')
        return bob_decrypt_error, eve_decrypt_error
```

- After the training we initialize the testing data set and call the test_case_gen() function to generate the test case.

```
msg_val,key_val=gen_data(n=512,msg_len=8,key_len=8)
msg_val = test_case_gen(512,8)
```

- This function reads the **test.txt** and converts each character into a binary representation and returns a array of message bits.

```
def test_case_gen(n=512,m=8):
    print(m ,n)
    f = open("test.txt", "r")
    test = f.read(512)
    print(type(test))
    print(test)

    #defing the list

    test_list=[]

    #convert into binary representation

    k = ' '.join('{0:08b}'.format(ord(x), 'b') for x in test[0])
    print(type(k),"---",k)
    for i in range(n):
        k = ' '.join('{0:08b}'.format(ord(x), 'b') for x in test[i])
        test_list_temp = []
        for j in range(m):

            digit= int(k[j])
            test_list_temp.append(digit)
        test_list.append(test_list_temp)
    test_array= np.array(test_list)
#     print("-------",test_list)
#     print(test_array.shape,"-----",test_array)
    return test_array
```

- We map all 0 bits to -1 and all 1 bits to 1.

```
msg_val[msg_val ==0]=-1
```

- We then call the **predict()** function to test from text file which returns the final bit arrays for B and E predicted output.

```
bob,eve=crypto_net.predict(msg_val,key_val)
```

- We map these decimal values to the bits comparing if it is nearer to -1 or 1 and then map them to 0 and 1 respectively.

```
 #BOB
  bob_1=np.subtract(bob,array_1)
  bob_1=np.absolute(bob_1)
  print(bob_1,"bob_1")
  bob_0=np.subtract(bob,array_0)
  print(bob_0,"bob_0")
  bob_0=np.absolute(bob_0)

  bob_ans= np.subtract(bob_0,bob_1)
  bob_ans = bob_ans.clip(0)
  bob_ans[bob_ans>0] = 1

  #EVE
  eve_1=np.subtract(eve,array_1)
  eve_1=np.absolute(eve_1)
  eve_0=np.subtract(eve,array_0)
  eve_0=np.absolute(eve_0)

  eve_ans= np.subtract(eve_0,eve_1)
  eve_ans = eve_ans.clip(0)
  eve_ans[eve_ans>0] = 1

  #print
  msg_val=msg_val.clip(0)
  print(msg_val)

  print(bob_ans,"BOB")

  print(eve_ans,"EVE")
```

- We then evaluate the number of wrong bits for error prediction and convert the bits back into text.

```
    bob_wrong = np.add(bob,msg_val)
    BOB_wrong = np.count_nonzero(bob_wrong == 1)
    eve_wrong = np.add(eve,msg_val)
    EVE_wrong = np.count_nonzero(eve_wrong == 1)

    #print

    print(BOB_wrong,"no of wrong bits of bob")
    print(EVE_wrong,"no of wrong bits of eve")

    bob_ans_text= output(bob_ans)

    print(bob_ans_text,"------","BOB TEXT ANS")

    eve_ans_text= output(eve_ans)

    print(eve_ans_text,"------","EVE TEXT ANS")
```

- The convertion functions used are :

```
def int2bytes(i):
    hex_string = '%x' % i
    n = len(hex_string)
    return binascii.unhexlify(hex_string.zfill(n + (n & 1)))

def text_from_bits(bits, encoding="ISO-8859-1", errors='surrogatepass'):
    n = int(bits, 2)
    return int2bytes(n).decode(encoding, errors)
```