PROJECT REPORT

Secure Communication With Adversarial Neural Cryptography

Harsh Mishra
Roll number- 19049
3rd year BS-MS

CONTENTS

ABSTRACT

For several years, encryption has been the standard method of establishing a secure connection. It is safe, computationally efficient, and practically every platform supports it. However, one disadvantage of encryption is that an encrypted message has no meaning. It's a jumble of meaningless bits and characters that no human can decipher. It encrypts and decrypts messages using ciphers that are human-readable. Each cipher is, more precisely, an image with the message hidden within it. The message (sentence) from the new image can be extracted by the other component of the network.

Cryptography is a broad term that refers to algorithms and protocols that secure the confidentiality and integrity of data. Programs or Turing machines are commonly used to describe cryptographic systems. Attackers are also defined in these terms, including bounds on their complexity (e.g., polynomial time) and success rates (e.g., limited to a negligible probability). If a mechanism fulfills its aim against all attackers, it is said to be secure. For example, if no attacker can extract information about plaintexts from ciphertexts, an encryption scheme is said to be secure.

We concentrate on ensuring secrecy properties in a multiagent system, and we express those properties in terms of an opponent. As an example, a system could consist of two neural networks named Alice and Bob, with the goal of limiting what a third neural network named Eve learns via eavesdropping on Alice and Bob's conversation. We train these neural networks end-to-end, adversarially, rather than prescribing specific cryptographic techniques. We show that neural networks can learn how to execute encryption and decryption in various ways, as well as how to apply these operations selectively to achieve confidentiality goals.

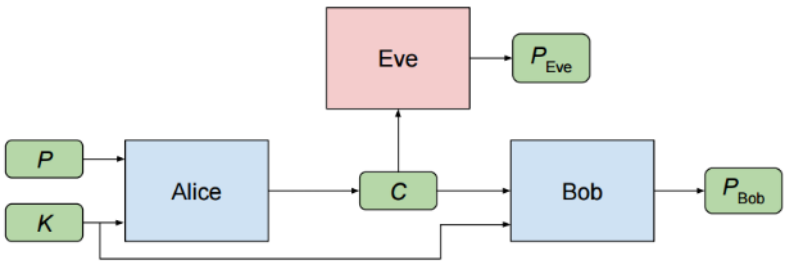
PROBLEM STATEMENT

Let's consider there are 2 persons, Alice and Bob, who want to converse safely over a public channel, and Eve is an attacker listening in. We can assume that Alice and Bob share a secret key for symmetric encryption. Eve can read all of Alice's encrypted messages to Bob, but she doesn't have access to the secret key.

The goal is for Alice to send a message to Bob and Eve (who is attempting to hack into Alice and Bob's communication) to be unable to read it. Alice takes a message (plain text) and encrypts it into ciphertext using the key she shares with Bob. This ciphertext is transferred to Bob, who decrypts the ciphertext into the plain text after consuming the communication and key. Eve intercepts the cipher text's transmission and tries to decrypt the plaintext without knowing the keys or, in some situations, the encryption technique.Our task is to know if neural networks can learn to safeguard information from other neural networks by using secret keys.

BACKGROUND

P is a single confidential message that Alice wishes to send to Bob. Alice receives the message P as an input. Alice generates C as a result of processing this input. (The letters "P" and "C" stand for "plaintext" and "ciphertext,") Bob and Eve each get C, process it, and try to locate P. P_{Bob} and P_{Eve} denote what they compute. Alice and Bob have an advantage over Eve because they have the same secret key, K. K is treated as a third input for Alice and Bob.



We assume that Alice and Bob share a secret key for symmetric encryption. Eve can read all of Alice's encrypted messages to Bob, but she doesn't have access to the secret key.

SETUP

I created three neural networks: Alice which is the encryption algorithm, Bob which is the decryption algorithm, and Eve which is the attacker. These three networks are quite similar.

Alice neural network-

The message and key vectors are concatenated into one long vector by Alice. This is then sent through a single fully-connected dense layer. It then goes through a standard convolutional setup, which takes the concatenated vector and runs it through a series of 1D convolution filters before producing an N-length vector. This is the C transmission that Bob receives.

```
Model: "alice"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 16)]              0
input_2 (InputLayer)         [(None, 16)]              0
concatenate (Concatenate)    (None, 32)                0
dense (Dense)                (None, 32)                1056
activation (Activation)      (None, 32)                0
reshape (Reshape)            (None, 32, 1)             0
conv1d (Conv1D)              (None, 32, 2)             10
activation_1 (Activation)    (None, 32, 2)             0
conv1d_1 (Conv1D)            (None, 16, 4)             20
activation_2 (Activation)    (None, 16, 4)             0
conv1d_2 (Conv1D)            (None, 16, 4)             20
activation_3 (Activation)    (None, 16, 4)             0
conv1d_3 (Conv1D)            (None, 16, 1)             5
activation_4 (Activation)    (None, 16, 1)             0
flatten (Flatten)            (None, 16)                0
=================================================================
Total params: 1,111
Trainable params: 1,111
Non-trainable params: 0
_________________________________________________________________
```

Bob neural network-

The only difference between Bob's and Alice's networks is that his input is the concatenation of the communication(ciphertext) and the key.

```
Model: "bob"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
input_3 (InputLayer)         [(None, 16)]              0         
_________________________________________________________________
input_4 (InputLayer)         [(None, 16)]              0         
_________________________________________________________________
concatenate_1 (Concatenate)  (None, 32)                0         
_________________________________________________________________
dense_1 (Dense)              (None, 32)                1056      
_________________________________________________________________
activation_5 (Activation)    (None, 32)                0         
_________________________________________________________________
reshape_1 (Reshape)          (None, 32, 1)             0         
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 32, 2)             10        
_________________________________________________________________
activation_6 (Activation)    (None, 32, 2)             0         
_________________________________________________________________
conv1d_5 (Conv1D)            (None, 16, 4)             20        
_________________________________________________________________
activation_7 (Activation)    (None, 16, 4)             0         
_________________________________________________________________
conv1d_6 (Conv1D)            (None, 16, 4)             20        
_________________________________________________________________
activation_8 (Activation)    (None, 16, 4)             0         
_________________________________________________________________
conv1d_7 (Conv1D)            (None, 16, 1)             5         
_________________________________________________________________
activation_9 (Activation)    (None, 16, 1)             0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 16)                0         
=================================================================
Total params: 1,111
Trainable params: 1,111
Non-trainable params: 0
```

Eve neural network-

Eve's network resembles Bob and Alice's in many ways. Her contribution, however, is limited to communication C(ciphertext). Prior to the usual convolutional design, Eve contains an additional fully-connected dense layer, giving her a greater chance of decrypting C since it doesn’t have access to the secret key.

```
Model: "eve"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_5 (InputLayer)         [(None, 16)]              0

dense_2 (Dense)              (None, 32)                544

activation_10 (Activation)   (None, 32)                0

dense_3 (Dense)              (None, 32)                1056

activation_11 (Activation)   (None, 32)                0

reshape_2 (Reshape)          (None, 32, 1)             0

conv1d_8 (Conv1D)            (None, 32, 2)             10

activation_12 (Activation)   (None, 32, 2)             0

conv1d_9 (Conv1D)            (None, 16, 4)             20

activation_13 (Activation)   (None, 16, 4)             0

conv1d_10 (Conv1D)           (None, 16, 4)             20

activation_14 (Activation)   (None, 16, 4)             0

conv1d_11 (Conv1D)           (None, 16, 1)             5

activation_15 (Activation)   (None, 16, 1)             0

flatten_2 (Flatten)          (None, 16)                0

=================================================================
Total params: 1,655
Trainable params: 1,655
Non-trainable params: 0
```

With this, our initial setup of three networks is completed.

What is Adversarial training?

An approach aimed at influencing a machine learning model to make a wrong prediction is known as an adversarial attack. Here, training proceeds in an adversarial way, we train Bob and Alice for a while until they can effectively communicate, then we train Eve for a while until she can decode the message. Then we retrain Bob and Alice, who find out how to decrypt Eve's current method of decryption. Then we retrain Eve, who decrypts the upgraded encryption, and so on. After a period, Bob and Alice's encryption becomes too difficult for Eve to decipher.

LOSS FUNCTIONS

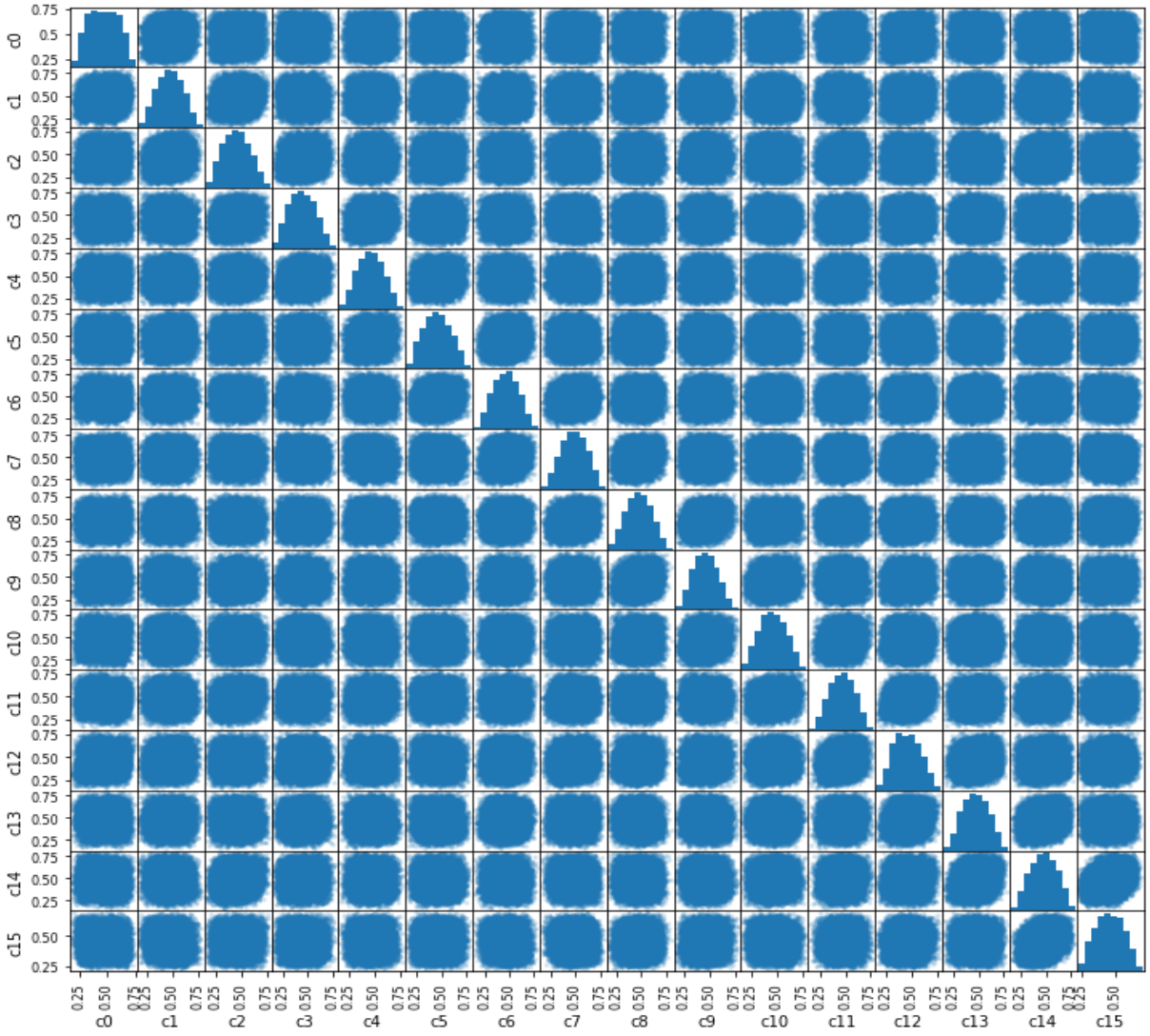
The absolute difference between the message vector P and Eve's attempt to decrypt it P_{Eve} is Eve's loss function. To update Eve's weights, this loss is backpropagated across the weights in the Eve network. As given in the code implementation part, the loss for Eve is just the L1 distance between ainput0 and eoutput. The sum of all the bits in the message is computed. Per-example loss is the quantity inside the K.mean() function. We average the results from the entire mini-batch.

```
eveloss = K.mean(  K.sum(K.abs(ainput0 - eveout), axis=-1)  )
```

The loss of Alice-Bob communication depends on Bob's reconstruction, but Eve's ability to decrypt the message is also reliant on Eve's capability. Eve shouldn't be able to guess any better than at random, thus she'll get half of the parts right on average. When Eve accurately guesses a few of the bits, the square term ensures that the error grows enormous. Alice is forced to respond to attackers more quickly and change its encryption strategy as a result of this. Overall, Alice and Bob's combined loss(abeloss) is given as:

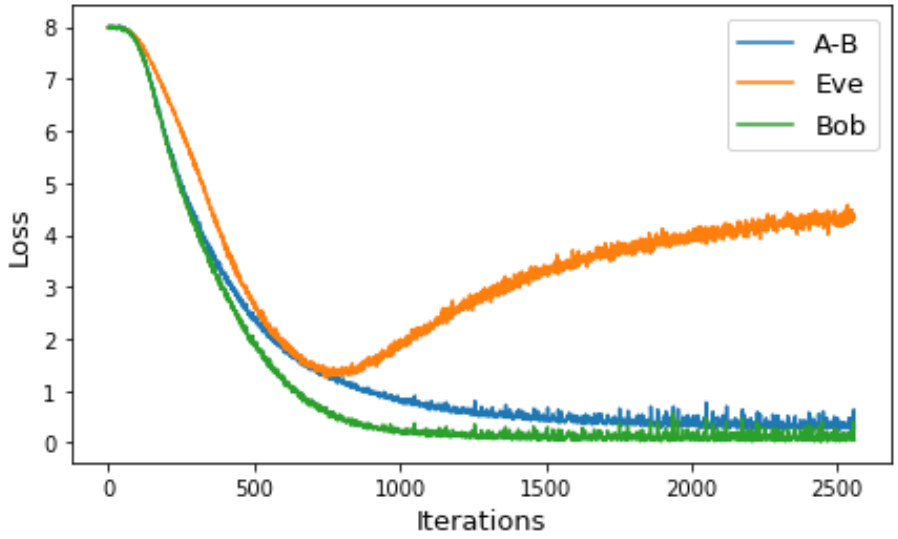
```
abeloss = bobloss + K.square(m_bits/2 - eveloss)/( (m_bits//2)**2 )
```

Correlation is the term used to describe the relationship between two variables. A scatter plot is often made up of a vast amount of data. The higher the correlation between the two variables, or the stronger the association, the closer the data points come to forming a straight line when plotted. The columns of the synthetic data which was created using random function were considered to find correlation and that gave us the below scatter matrix-



RESULTS

I kept track of both the values of the loss of Alice-Bob combined and Eve's loss and also looked at Bob's capacity to decipher Alice's messages. As training progressed, the plot for loss values is as below for model Crypto1 by Google, check the “models” folder in the zip file-

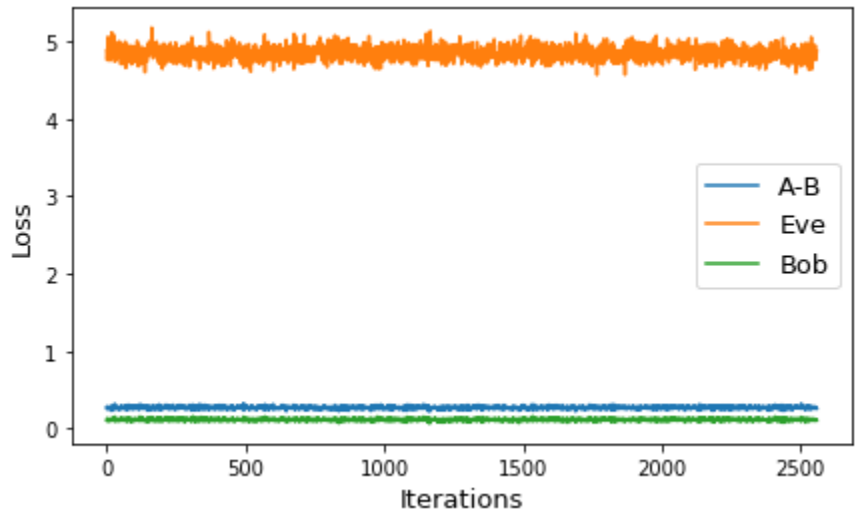


As can be clearly seen from the plot, all three loss values start at 8, implying that neither Eve nor Bob outperforms random guessing. Eve's loss accelerates up to a point, at which point Alice is able to figure out an encryption strategy that Bob can decipher but which deceives Eve. Eve's loss does not return to 8, indicating that she does somewhat better than random guessing. The evaluation was done for 10,000 random messages which gave the result as below-

```
Bob % correct:  88.54 %
Eve % correct:  0.21 %
```

Bob is able to decrypt around 90% of the messages he gets, however Eve is unable to recover the entire message in its current state. Eve's performance would improve if the Alice-Bob model was frozen and we continued to train her.It's critical to keep in mind that Eve still outperforms random guesses. The chance of accurately guessing the entire message under the condition of independent

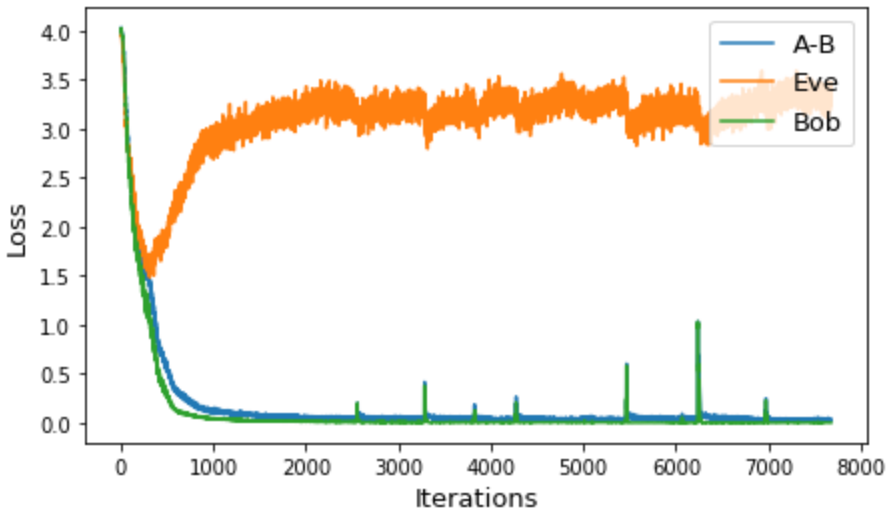
guesses for each ciphertext bit is $1/2^{16} = 0.0015\%$ which is less than what we achieved. When Alice-Bob is frozen, the performance of Eve is as below-




```
Bob % correct:  88.23 %
Eve % correct:  0.24 %
```

When the trained form of Alice-Bob is kept constant, we discover that Eve's advantage does not improve. The loss is rather consistent, implying that Alice's encryption technique does not need to be modified on a regular basis.

For 8000 iterations, the training progress is as below for model Crypto2 (for which Bob was incredibly accurate while Eve failed to decrypt the messages). by Google-




```
Bob % correct:  100.0 %
Eve % correct:  1.37 %
```

The decryption algorithm's high level of precision makes this a viable basis for creating a toy-model cryptosystem that can function on human-readable characters.

PROOF OF CONCEPT(AI encryption)

On 8-bit messages/keys and 8-dimensional ciphertexts, we created and trained similar neural configurations. A dense layer was added after the sequence of 1D convolutions, and the codings were allowed to be between 1 and -1(using tanh activation function), in addition to the bit size.

Algorithm for Encryption:

- ☐ Convert a string of characters to binary.
- ☐ To get a block of 8 bits per character, pad the binary char encoding string with 3 random bits.
- ☐ Through the encryption network Bob, send 8-bit blocks of the string.
- ☐ The result is an 8-dim float32 vector with a total of 8*32=256 bits per character.
- ☐ Concatenate the binary representations of each component in the 8-dim float vector to create a binary string. This is how the data is encoded.

To view what the message might look like, change it to a character string. Each character will be a 5 character (remove 3 bits for padding).The reverse of the encryption method is used for decryption.

In our code, the plaintext “harsh” has the binary representation as below:

```
10000111001000000101000111111001000000111
harsh
```

However, its ciphertext encoding is far worse. It looks like below when the 5-bit map is used to map it to human-readable output.

```
?jzs:nc !bvz:!we?k u,fah?zhr?nzp? pn!tl ?xwa::f?!jm:,  i,df??sso?fm.,vui, oh:slk! ap?fwy.ifu?nj!??wh!j.e?,,d:
n,!nl:!vht.:e!?srl?,s ?xmi!pwn:myk!m!r!wjb?q :!egs
```

Bob is able to decode the binary ciphertext successfully, whereas Eve's code-breaking attempt yielded-

```
kkikk
```

This was just a proof of concept, there are a number of alternative ways to use the trained Alice and Bob models to build a secure cryptosystem.

CONCLUSION

We show how neural networks can learn to safeguard communications in this research. The learning does not necessitate defining a specific collection of cryptographic algorithms or stating how to use them; it is just dependent on a secrecy specification provided by the training objectives. In this case, we use neural networks to model attackers; alternate models may be enabled through reinforcement learning. We show that neural networks can learn how to execute encryption and decryption in various ways, as well as how to apply these operations selectively to achieve confidentiality goals.

Through this project, I attempt to explore how neural networks could be efficient for cryptography. I introduce a symmetric encryption neural cryptography algorithm that not only ensures secrecy but also satisfies the three security standards (Confidentiality, Integrity, and Availability). Rather than concentrating on cryptanalysis (which would have strengthened Eve), I concentrated on increasing the confidentiality of the message sent between Alice and Bob. The goal is to reduce the error rate of the algorithm by adjusting its hyperparameters, as well as to construct the protocol using network sockets and simulate communication between Alice and Bob.

CODING IMPLEMENTATION

Importing essential libraries that will help us in building the code-

```
import ctypes
import gmpy2
from gmpy2 import mpz
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sys

from keras import backend as K
from keras.models import Model
from keras.engine.input_layer import Input
from keras.layers.core import Activation, Dense
from keras.layers import Flatten, Reshape
from keras.layers.convolutional import Conv1D
from keras.layers.merge import concatenate
from tensorflow.keras.optimizers import Adam, RMSprop
from keras.models import load_model
```

2 input vectors will be required by the *Alice network*: the encrypted message and the secret key. These are concatenated and given to a dense layer along axis=1. The signal is then processed through four Conv1D layers to produce the output.

```
#Alice network

ainput0 = Input(shape=(m_bits,)) #message
ainput1 = Input(shape=(k_bits,)) #key
ainput = concatenate([ainput0, ainput1], axis=1)

adense1 = Dense(units=(m_bits + k_bits))(ainput)
adense1a = Activation('tanh')(adense1)

areshape = Reshape((m_bits + k_bits, 1,))(adense1a)

aconv1 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(areshape)
aconv1a = Activation('tanh')(aconv1)
aconv2 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(aconv1a)
aconv2a = Activation('tanh')(aconv2)
aconv3 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(aconv2a)
aconv3a = Activation('tanh')(aconv3)
aconv4 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(aconv3a)
aconv4a = Activation('tanh')(aconv4)

aflat = Flatten()(aconv4a)
aoutput = Dense(units=c_bits, activation='tanh')(aflat) #ciphertext

alice = Model([ainput0, ainput1], aoutput, name='alice')
#alice.summary()
```

Bob and Eve neural networks-

The Dense layer's output will be two-dimensional, with mini-batch instances arranged along the first axis. Before being supplied to a Conv1D layer, this must be reshaped to (batch size, m bits + k bits, 1), introducing a third dimension that would typically correspond to the number of channels (here 1). The Bob network's architecture is identical to Alice's, with the exception that input0 now represents the ciphertext rather than the plaintext while the Eve network has 2 DENSE layers which act.

```
#Bob network

binput0 = Input(shape=(c_bits,)) #ciphertext
binput1 = Input(shape=(k_bits,)) #key
binput = concatenate([binput0, binput1], axis=1)

bdense1 = Dense(units=(c_bits + k_bits))(binput)
bdense1a = Activation('tanh')(bdense1)

breshape = Reshape((c_bits + k_bits, 1,))(bdense1a)

bconv1 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(breshape)
bconv1a = Activation('tanh')(bconv1)
bconv2 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(bconv1a)
bconv2a = Activation('tanh')(bconv2)
bconv3 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(bconv2a)
bconv3a = Activation('tanh')(bconv3)
bconv4 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(bconv3a)
bconv4a = Activation('tanh')(bconv4)

bflat = Flatten()(bconv4a)
boutput = Dense(units=m_bits, activation='sigmoid')(bflat) #decrypted message

bob = Model([binput0, binput1], boutput, name='bob')
#bob.summary()


#Eve network

einput = Input(shape=(c_bits,)) #ciphertext only

edense1 = Dense(units=(c_bits + k_bits))(einput)
edense1a = Activation('tanh')(edense1)
edense2 = Dense(units=(m_bits + k_bits))(edense1a)
edense2a = Activation('tanh')(edense2)

ereshape = Reshape((m_bits + k_bits, 1,))(edense2a)

econv1 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(ereshape)
econv1a = Activation('tanh')(econv1)
econv2 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(econv1a)
econv2a = Activation('tanh')(econv2)
econv3 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(econv2a)
econv3a = Activation('tanh')(econv3)
econv4 = Conv1D(filters=4, kernel_size=kersize, strides=1, padding=pad)(econv3a)
econv4a = Activation('tanh')(econv4)

eflat = Flatten()(econv4a)
eoutput = Dense(units=m_bits, activation='sigmoid')(eflat) #code break attempt

eve = Model(einput, eoutput, name='eve')
#eve.summary()
```

For loss and optimizers as discussed previouslym we take help of the K.mean function. We utilise RMSprop with a default learning rate of 0.001 according to the optimizer.

```
#Linking inputs to outputs

aliceout = alice([ainput0, ainput1])
bobout = bob( [aliceout, binput1] )# bob sees ciphertext AND key
eveout = eve( aliceout )# eve doesn't see the key, only the cipher

# Loss for Eve
eveloss = K.mean(  K.sum(K.abs(ainput0 - eveout), axis=-1)  )

# Loss for Alice-Bob communication

bobloss = K.mean(  K.sum(K.abs(ainput0 - bobout), axis=-1)  )
abeloss = bobloss + K.square(m_bits/2 - eveloss)/( (m_bits//2)**2 )

# Optimizer and compilation

abeoptim = Adam()#RMSprop(lr=0.0015)
eveoptim = Adam()#RMSprop(lr=0.0015) #default 0.001


# Build and compile the ABE(alice-bob-eve) model, used for training Alice-Bob networks

abemodel = Model([ainput0, ainput1, binput1], bobout, name='abemodel')
abemodel.add_loss(abeloss)
abemodel.compile(optimizer=abeoptim)


# Build and compile the EVE model, used for training Eve net (with Alice frozen)

alice.trainable = False
evemodel = Model([ainput0, ainput1], eveout, name='evemodel')
evemodel.add_loss(eveloss)
evemodel.compile(optimizer=eveoptim)
```

Creating synthetic data:

```
n_examples = 10000
showAll = True

coord_indeces = np.array([
    [ 0, 1, 2, 4],
    [ 5, 6, 7, 14]
])

m_batch = np.random.randint(0, 2, m_bits * n_examples).reshape(n_examples, m_bits)
k_batch = np.random.randint(0, 2, m_bits * n_examples).reshape(n_examples, m_bits)
m_enc = alice.predict([m_batch, k_batch])

if showAll:
    n_cols = 4
    n_rows = m_enc.shape[1] // n_cols
else:
    n_cols = coord_indeces.shape[1]
    n_rows = coord_indeces.shape[0]

plt.figure(figsize=(8, int(8.0/n_cols * n_rows)))
for row in range(n_rows):
    for col in range(n_cols):
        i = row * n_cols + col
        plt.subplot(n_rows, n_cols, i + 1)
        if showAll:
            plt.title("Coord " + str(i), fontsize=14)
            plt.hist(m_enc[:, i], bins=20, density=True)
        else:
            plt.title("Coord " + str(coord_indeces[row, col]), fontsize=12)
            plt.hist(m_enc[:, coord_indeces[row, col]], bins=20, density=True)
```

Training:

```
n_epochs = 30
batch_size = 256
n_batches = m_train // batch_size

abecycles = 1
evecycles = 2

epoch = 0
print("Training for", n_epochs, "epochs with", n_batches, "batches of size", batch_size)

while epoch < n_epochs:
    abelosses0 = []
    boblosses0 = []
    evelosses0 = []
    for iteration in range(n_batches):

        # Train the network

        alice.trainable = True
        for cycle in range(abecycles):
            # Select a random batch of messages, and a random batch of keys

            m_batch = np.random.randint(0, 2, m_bits * batch_size).reshape(batch_size, m_bits)
            k_batch = np.random.randint(0, 2, k_bits * batch_size).reshape(batch_size, k_bits)
            loss = abemodel.train_on_batch([m_batch, k_batch, k_batch], None)

        abelosses0.append(loss)
        abelosses.append(loss)
        abeavg = np.mean(abelosses0)
```

```
        # Evaluate Bob's ability to decrypt a message
        m_enc = alice.predict([m_batch, k_batch])
        m_dec = bob.predict([m_enc, k_batch])
        loss = np.mean(  np.sum( np.abs(m_batch - m_dec), axis=-1)  )
        boblosses0.append(loss)
        boblosses.append(loss)
        bobavg = np.mean(boblosses0)

        # Train the EVE network

        alice.trainable = False
        for cycle in range(evecycles):
            m_batch = np.random.randint(0, 2, m_bits * batch_size).reshape(batch_size, m_bits)
            k_batch = np.random.randint(0, 2, k_bits * batch_size).reshape(batch_size, k_bits)
            loss = evemodel.train_on_batch([m_batch, k_batch], None)

        evelosses0.append(loss)
        evelosses.append(loss)
        eveavg = np.mean(evelosses0)

        if iteration % max(1, (n_batches // 100)) == 0:
            print("\rEpoch {:3}: {:3}% | abe: {:2.3f} | eve: {:2.3f} | bob: {:2.3f}".format(
                epoch, 100 * iteration // n_batches, abeavg, eveavg, bobavg), end="")
            sys.stdout.flush()

    print()
    epoch += 1

print('Training finished.')
```

Output-

```
Training for 30 epochs with 256 batches of size 256
Epoch   0:  99% | abe: 2.547 | eve: 2.556 | bob: 2.374
Epoch   1:  99% | abe: 1.028 | eve: 1.749 | bob: 0.702
Epoch   2:  99% | abe: 0.315 | eve: 2.297 | bob: 0.127
Epoch   3:  99% | abe: 0.148 | eve: 2.804 | bob: 0.057
Epoch   4:  99% | abe: 0.106 | eve: 2.954 | bob: 0.036
Epoch   5:  99% | abe: 0.084 | eve: 3.048 | bob: 0.026
Epoch   6:  99% | abe: 0.069 | eve: 3.136 | bob: 0.020
Epoch   7:  99% | abe: 0.061 | eve: 3.183 | bob: 0.016
Epoch   8:  99% | abe: 0.050 | eve: 3.263 | bob: 0.013
Epoch   9:  99% | abe: 0.052 | eve: 3.242 | bob: 0.013
Epoch  10:  99% | abe: 0.055 | eve: 3.164 | bob: 0.010
Epoch  11:  99% | abe: 0.050 | eve: 3.203 | bob: 0.009
Epoch  12:  99% | abe: 0.059 | eve: 3.205 | bob: 0.017
Epoch  13:  99% | abe: 0.053 | eve: 3.137 | bob: 0.006
Epoch  14:  99% | abe: 0.052 | eve: 3.217 | bob: 0.011
Epoch  15:  99% | abe: 0.047 | eve: 3.225 | bob: 0.007
Epoch  16:  99% | abe: 0.051 | eve: 3.243 | bob: 0.013
Epoch  17:  99% | abe: 0.047 | eve: 3.213 | bob: 0.006
```

...

Model evaluation:

```
n_examples = 10000

m_batch = np.random.randint(0, 2, m_bits * n_examples).reshape(n_examples, m_bits)
k_batch = np.random.randint(0, 2, m_bits * n_examples).reshape(n_examples, m_bits)

m_enc = alice.predict([m_batch, k_batch])
#m_enc = np.round(m_enc, 3)
m_dec = (bob.predict([m_enc, k_batch]) > 0.5).astype(int)
m_att = (eve.predict(m_enc) > 0.5).astype(int)

bdiff = np.abs(m_batch - m_dec)
bsum = np.sum(bdiff, axis=-1)
ediff = np.abs(m_batch - m_att)
esum = np.sum(ediff, axis=-1)

print("Bob % correct: ", 100.0*np.sum(bsum == 0) / n_examples, '%')
print("Eve % correct: ", 100.0*np.sum(esum == 0) / n_examples, '%')
```

Encryption powered by artificial intelligence (PoC)

```
key = np.array([[0,0,0,0,0,0,0,0]])
m = 'harsh'

m_bin, _ = encstr(m, block_padding=3)
m_bin_len = len(m_bin)
print(m_bin, m_bin_len)

ciphertext = ""
for i in range(m_bin_len // m_bits):
    # read blocks of size m_bits
    binblockstr = m_bin[m_bits*i : m_bits*i + m_bits]
    binblock = np.array(list(binblockstr)).astype(np.int8).reshape(1, m_bits)

    floatVector = alice.predict([binblock, key])
    #print(np.round(floatVector,3))

    # convert each coordinate of the cipher (float) vector to binary
    # and construct the binary ciphertext
    for j in range(c_bits):
        ciphertext = ciphertext + float_to_binary(floatVector[0][j])
        #print(float_to_binary(floatVector[0][j]))

#print(ciphertext, len(ciphertext)) # ciphertext in binary
print(decstr(ciphertext, n=(len(ciphertext)//8), block_padding=3)) #ciphertext as characters

ciphertext_len = len(ciphertext)
plaintextbin = ""
for i in range(ciphertext_len // (c_bits*32)):
    # read the ciphertext in chunks of 32*c_bits bits, i.e one encoding at a time
    floatVectorbin = ciphertext[c_bits*32*i : c_bits*32*i + c_bits*32]
    #print(floatVectorbin)
    # convert the binary chunk to an 8-dim float vector (input for AI Bob)
    floatVector = np.zeros(c_bits, dtype=np.float32).reshape(1, c_bits)
```

```
    for j in range(len(floatVectorbin) // 32):
        floatValuebin = floatVectorbin[32*j : 32*j + 32]
        #print(floatValuebin)
        floatValue = binary_to_float(floatValuebin)
        floatVector[0][j] = floatValue
    #print(np.round(floatVector,3))

    charbinvector = list( (bob.predict([floatVector, key]) > 0.5)[0].astype(int) )
    for j in range(len(charbinvector)):
        plaintextbin = plaintextbin + str(charbinvector[j])

print(plaintextbin)

m_dec = ""
for i in range(len(plaintextbin) // m_bits):
    strbin = plaintextbin[m_bits*i : m_bits*i + m_bits]
    m_dec = m_dec + decstr(strbin, len(strbin)//m_bits, block_padding=3)

print(m_dec)
```

REFERENCES

1. Learning to Protect Communications with Adversarial Neural Cryptography by Martín Abadi, David G. Andersen (Google Brain).
2. https://towardsdatascience.com/neural-cryptography-7733f18184f3
3. https://keras.io/
4. An Adversarial Neural Cryptography Approach to Integrity Checking: Learning to Secure Data Communications by IEEE.