A

Project Report

On

Performance Comparison of Autoencoders with Hamming Codes

Subject: 5G Communication and Network (EC431)

Submitted by

Ms. Sakshi Somani (202441002)

Ms. Ila Chaudhary (202471007)

Abstract: In this project, we will learn how to implement an end-to-end digital communication system as an AutoEncoder and compared its performance with a (n,k) Hamming Code. We analyze the performance of an autoencoder based modeling of a communication system as opposed to traditional modeling, where designing signal alphabet at transmitter (Tx) and detection algorithms at receiver (Rx) are based on a given mathematical/statistical channel/system model.We simulate the performance of an AutoEncoder based communication link in the presence of Additive White Gaussian Noise (AWGN), where Tx sends one out of M message/information symbols per n channel uses through a noisy channel and the Rx estimates the transmitted symbols through noisy observations. The goal is to learn a signalling alphabet/constellation scheme that is robust with respect to the noise introduced by the channel at Tx.

Keywords : AutoEncoder, AWGN, Hamming Code.

Abstract

INDEX -

1. Objective

The objective of this minor project is

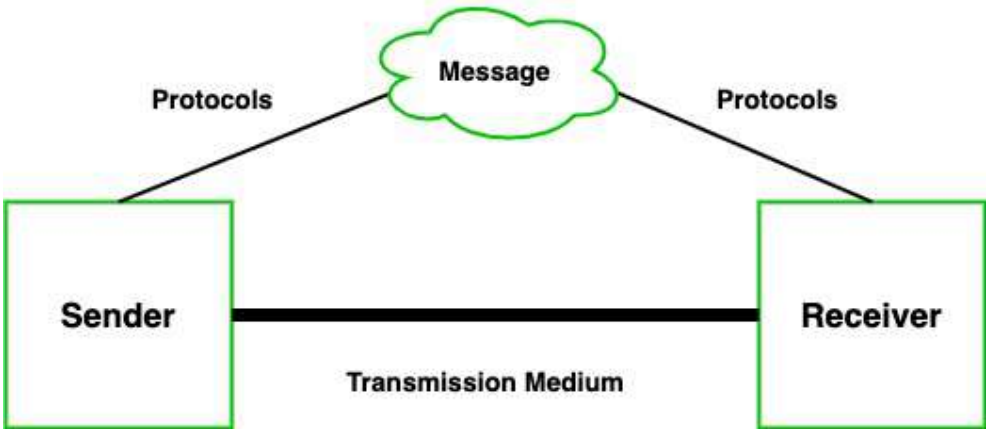
- To analyse the performance of an autoencoder based modeling of a communication system as opposed to traditional modeling, where designing signal alphabet at transmitter (Tx) and detection algorithms at receiver (Rx) are based on a given mathematical/statistical channel/system model.
- To simulate the performance of an AutoEncoder based communication link in the presence of Additive White Gaussian Noise (AWGN), where Tx sends one out of M message/information symbols per n channel uses through a 1+ cv noisy channel and the Rx estimates the transmitted symbols through noisy observations.
- To implement Hamming Code Encoder and Decoder
- To compare the BLER performance of AutoEncoder with Hamming Code for (n,k) = (7,4)
- To summarise implement an end-to-end digital communication system as an Autoencoder and compared its performance with a (n,k) Hamming Code.
- To study Constellation Learning for the lower level codes like (2,4) or (2,2) against Energy and Power Constraints

2. Encoding and Decoding Techniques

Encoding and **decoding** are fundamental processes in communication that ensure the accurate transmission and interpretation of messages between a sender and a receiver.

Encoding refers to the process of converting information from one system to another using a set of characters, symbols, or signs. This process involves structuring the message, idea, or information in a way that can be effectively transmitted to the receiver[1]. The encoder, or source, creates a clear and straightforward message that can be easily understood by the receiver.

Decoding, on the other hand, is the process by which the receiver interprets the encoded message using their experiences and intellect[1]. The goal is to ensure that the message is clear, factual, and meaningful, allowing the receiver to comprehend and respond appropriately.



Importance of Encoding and Decoding

Encoding and decoding are crucial in communication for several reasons:

1. **Maintaining Data Integrity**: When transmitting data between different systems, especially in heterogeneous environments, encoding ensures that the data structure is preserved. This is important because different systems may have varying architectures and storage requirements.
2. **Handling Program Objects**: Encoding helps in converting program objects into a stream format suitable for transmission. This is necessary because absolute pointer values and storage requirements can vary between systems. Decoding then reconstructs these program objects from the received message data.

3. Hamming Codes

Hamming code is an error correction system that can detect and correct errors when data is stored or transmitted. It requires adding additional parity bits with the data. It is commonly used in error correction code (ECC) RAM. It's named after its inventor, Richard W. Hamming.

Whenever data is transmitted or stored, it's possible that the data may become corrupted. This can take the form of bit flips, where a binary 1 becomes a 0 or vice versa. Error correcting codes seek to find when an error is introduced into some data. This is done by adding parity bits, or redundant information, to the data.

If enough parity data is added, it enables forward error correction (FEC), where errors can be automatically fixed when read back. FEC can increase the data transmission rate for noisy channels by reducing the amount of necessary retransmits.

Hamming code uses a block parity mechanism. The data is divided into blocks, and parity is added to the block. Hamming code can correct single-bit errors and detect the presence of two-bit errors in a data block.

The amount of parity data added to Hamming code is given by the formula $2^p \geq d + p + 1$, where *p* is the number of parity bits and *d* is the number of data bits. For example, if you wanted to transmit 7 data bits, the formula would be $2^4 \geq 7 + 4 + 1$, so 4 parity bits are required.

The type of error correcting is given as the name of the algorithm, the total number of bits in the block and then the number of effective data bits. Therefore, Hamming encoding with 4 data bits and 3 parity bits for a total of 7 total bits is given as Hamming (7,4).

Standard Hamming code can only detect and correct a single-bit error. If two bits are in error, it is possible that the two errors will look like a single-bit error. To account for that, an additional overall parity bit can be added to reliably detect errors in two bits. This is known as *single error correction/double error detection* (SECDED).

The generating matrix (G) and the check matrix (H) for an (n,k) Hamming Code are defined given only the number of parity bits (M). Let M be the number of parity bits, then $n = 2^M-1$, and k = n - M. For example, M=3 produces a (7,4) code.

```
>> M=3;              % M = the number of parity bits
>> [H,G] = hammgen(M)  % return two matrices for an (n,k) Hamming code
```

H =

1	0	0	1	0	1	1
0	1	0	1	1	1	0
0	0	1	0	1	1	1

G =

1 1 0 1 0 0 0

0 1 1 0 1 0 0

1 1 1 0 0 1 0

1 0 1 0 0 0 1

Given either H or G as above, the other matrix (G or H) can be found with one command:

>> H = gen2par(G) % given G, find H

H =

1 0 0 1 0 1 1

0 1 0 1 1 1 0

0 0 1 0 1 1 1

Using the 4-bit message M = [0 1 0 1] and GF(2) arithmetic, the 7-bit codeword is created using generating matrix (G):

>> M = gf([0 1 0 1]); % Define a 4-bit message M

>> C = M * G % Create a 7-bit codeword from the message

C = GF(2) array.

Array elements = **1 1 0 0 1 0 1**

Any codeword C can be checked for errors using the check matrix H to give a 3-bit syndrome. In this case, the above codeword C has no errors.

>> H * C' % check for errors (find the syndrome) for the codeword C

ans = GF(2) array.

Array elements = 0 0 0

The minimum Hamming distance between any two codewords from the generating polynomial (G) is found as:

>> d = gfweight(G) % expect d=3 for single-bit error correction

d = 3

Correcting an error involves identifying a row in the syndrome table from the error remainer. Consider the single bit error E(x) and find the corresponding remainer (R) as if E(x) alone was the codeword:

>> E = [0 0 0 1 0 0 0]; % E(x) = error pattern

>> R = H * E' % R(x) = syndrome for the error

R = GF(2) array

Array elements = 1 1 0

Coding and Decoding with the Hamming Code

The Hamming code is expected to correct single bit errors. The **encode( )** and **decode( )** commands automatically incorporate error correction when used with the Hamming code.

While the **encode( )** and **decode( )** commands are very versatile, their default parameters define cyclic Hamming codes with binary inputs.

For example, the message [1 0 1 1] from the above examples is coded directly into a (7,4) Hamming code as:

>> M = [1 0 1 1]; % M(x) = four message bits

>> C = encode(M,7,4) % C(x) = A (7,4) Hamming codeword

C = 1 0 0 1 0 1 1

The resulting 7-bit codeword is decoded to reveal the original message as:

>> decode(C,7,4) % Decode the (7,4) Hamming codeword

ans= 1 0 1 1

The decode operation appears trivial when there are no errors as the message can be seen in the last four bits of this systematic codeword; however, the same message bits are returned even if a single bit error is added:

>> E = [0 0 0 1 0 0 0]; % E(x) = a single bit error

>> CC = bitxor(C,E) % Add the single bit error to the codeword

CC = 1 0 0 0 0 1 1

>> decode(CC,7,4) % Decode the errored codeword

ans= 1 0 1 1

The correct original message bits have been returned despite the presence of a bit error.

4. AutoEncoders

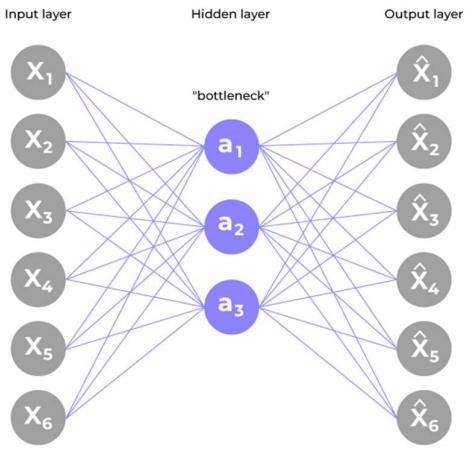
Autoencoders emerge as a fascinating subset of neural networks, offering a unique approach to unsupervised learning. Autoencoders are an adaptable and strong class of architectures for the dynamic field of deep learning, where neural networks develop constantly to identify complicated patterns and representations. With their ability to learn effective representations of data, these unsupervised learning models have received considerable attention and are useful in a wide variety of areas, from image processing to anomaly detection.

What are Autoencoders?

Autoencoders are a specialized class of algorithms that can learn efficient representations of input data with no need for labels. It is a class of artificial neural networks designed for unsupervised learning. Learning to compress and effectively represent input data without specific labels is the essential principle of an automatic decoder. This is accomplished using a two-fold structure that consists of an encoder and a decoder. The encoder transforms the input data into a reduced-dimensional representation, which is often referred to as “latent space” or “encoding”. From that representation, a decoder rebuilds the initial input. For the network to gain meaningful patterns in data, a process of encoding and decoding facilitates the definition of essential features.

Architecture of Autoencoder in Deep Learning

The general architecture of an autoencoder includes an encoder, decoder, and bottleneck layer.



1. Encoder
    - Input layer take raw input data
    - The hidden layers progressively reduce the dimensionality of the input, capturing important features and patterns. These layer compose the encoder.

- The bottleneck layer (latent space) is the final hidden layer, where the dimensionality is significantly reduced. This layer represents the compressed encoding of the input data.

2. Decoder
    - The bottleneck layer takes the encoded representation and expands it back to the dimensionality of the original input.
    - The hidden layers progressively increase the dimensionality and aim to reconstruct the original input.
    - The output layer produces the reconstructed output, which ideally should be as close as possible to the input data.
3. The loss function used during training is typically a reconstruction loss, measuring the difference between the input and the reconstructed output. Common choices include mean squared error (MSE) for continuous data or binary cross-entropy for binary data.
4. During training, the autoencoder learns to minimize the reconstruction loss, forcing the network to capture the most important features of the input data in the bottleneck layer.

After the training process, only the encoder part of the autoencoder is retained to encode a similar type of data used in the training process. The different ways to constrain the network are: –

- **Keep small Hidden Layers:** If the size of each hidden layer is kept as small as possible, then the network will be forced to pick up only the representative features of the data thus encoding the data.
- **Regularization:** In this method, a loss term is added to the cost function which encourages the network to train in ways other than copying the input.
- **Denoising:** Another way of constraining the network is to add noise to the input and teach the network how to remove the noise from the data.
- **Tuning the Activation Functions:** This method involves changing the activation functions of various nodes so that a majority of the nodes are dormant thus, effectively reducing the size of the hidden layers.

5. Implementation and Results

PHY layer as AutoEncoder

- The fundamental idea behind this tutorial is to model Physical Layer as an AutoEncoder (AE).
- An AutoEncoder (AE) is an Artificial Neural Network (ANN) used to learn an efficient representation of data at an intermediate layer to reproduce the input at its output.
- We Interpret end to end communication link, i.e., Tx, channel, and Rx as a single Neural Network (NN) that can be trained as an AE which reconstructs its input at its output, as communication is all about reproducing/reconstructing messages transmitted by Tx at Rx faithfully in the presence of channel perturbations and Rx noise.

Steps:

Following are the steps that we follow in simulation:

- Define the hyper parameters of AE :
    - number of information symbols M, where each symbol carry kbits
    - number of channel uses n
    - snr in dB at which AE is being trained, which we call snr_train
- Define embeddings for each information symbol that is to be fed as an input to AE.
- Define training and testing data set by randomizing the label
- Define end to end AutoEncoder by using the already imported keras built in layers.

Tx is being implemented as a stack of two keras Dense layers one with ReLU activation and another with linear activation. The output of the second Dense layer is fed to a normalization layer which we implement using keras Lambda layer.

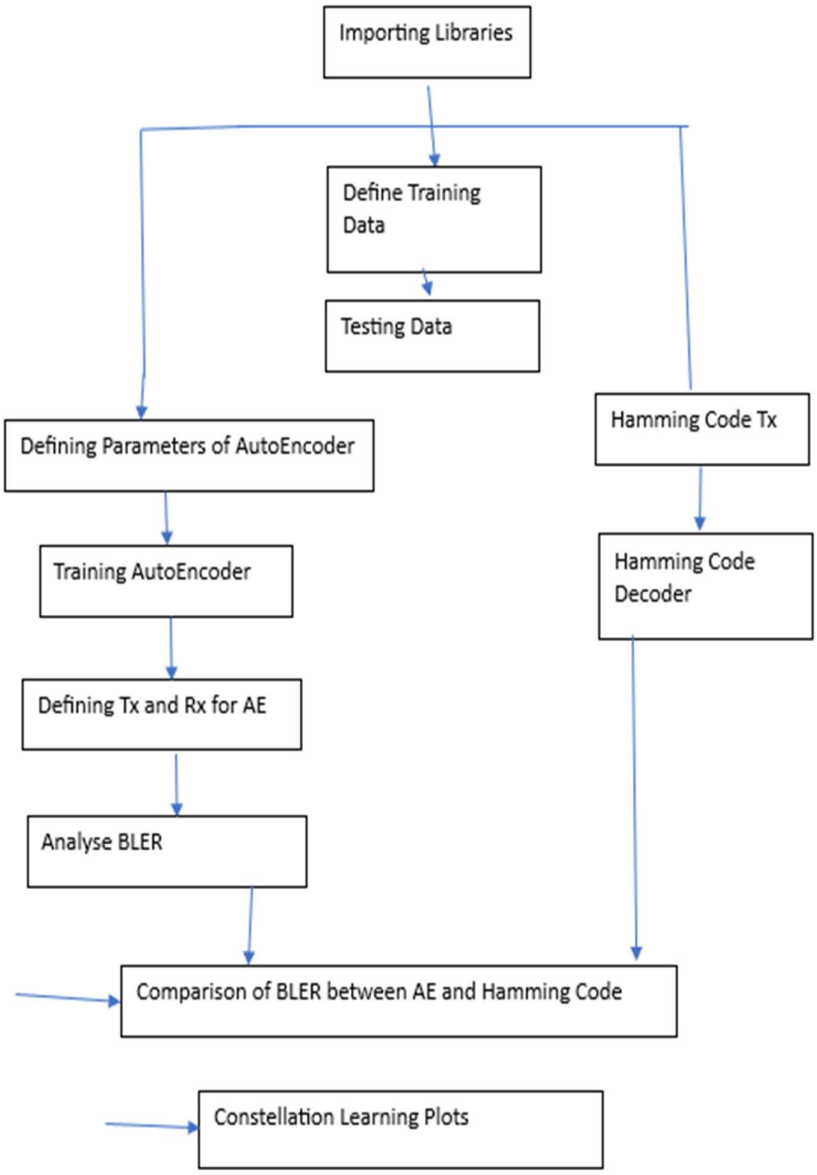
Channel is implemented as a single Noise layer with certain noise standard deviation, which is a function of both Rate of the code (R) and snr_train.

Rx also consists of two keras Dense layers, one with ReLU activation and the last one with softmax activation. The last layer must output probabilities (i.e., for a given received noisy vector v of dimension M, it outputs max aposteriori probability vector of dimension M, i.e., max(prob(w|v)) for any w belongs to transmitted oneshot embeddings)

Note: We choose different values of number of training (N) and testing samples (N_test) for constellation plots and for BlockErrorRate (BLER) plots.

Note: For (n,k) = (7,4), we use sklearn T-distributed Stochastic Neighbor Embedding (TSNE) to plot the learned constellation. Typically we use less number of N and N_test in such cases. For BLER plots we always go with high values of N, N_test

Implementation Flow



1. Importing Libraries

```
[ ] import os
    os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
    os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'


    # Importing necessary Numpy, Matplotlib, TensorFlow, Keras and scikit-learn modules
    # %matplotlib widget
    # %matplotlib inline
    import matplotlib.pyplot as plt
    import numpy as np
    import tensorflow as tf
    from keras.layers import Input, Dense, GaussianNoise, Lambda, BatchNormalization
    from keras.models import Model
    from keras.optimizers import Adam, SGD
    from keras import backend as be
```

2. Cloning 5G Toolkit to access 5G Toolkit

```
[ ] !git clone --branch website https://github.com/GigayasaWireless/toolkit5G.git
```

```
--2024-10-28 07:43:41--  https://github.com/GigayasaWireless/toolkit5G/archive/refs/heads/website.zip
Resolving github.com (github.com)... 140.82.113.4
Connecting to github.com (github.com)|140.82.113.4|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://codeload.github.com/GigayasaWireless/toolkit5G/zip/refs/heads/website [following]
--2024-10-28 07:43:41--  https://codeload.github.com/GigayasaWireless/toolkit5G/zip/refs/heads/website
Resolving codeload.github.com (codeload.github.com)... 140.82.112.9
Connecting to codeload.github.com (codeload.github.com)|140.82.112.9|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/zip]
Saving to: ‘toolkit5G.zip’

toolkit5G.zip           [              <=>  ] 267.64M  14.1MB/s    in 16s

2024-10-28 07:43:58 (16.4 MB/s) - ‘toolkit5G.zip’ saved [280643619]
```

3. Unzip 5G Toolkit

```
[ ] !unzip toolkit5G.zip
```

```
Archive:  toolkit5G.zip
417f1ca5b87c5a70793e6976d42f00491301ade8
replace toolkit5G-website/.buildinfo? [y]es, [n]o, [A]ll, [N]one, [r]ename:
```

4. Importing OS
   Importing sys and appending path to reach desired directory

```
[ ] import os
    os.listdir()

['.config',
 'test_GettingStarted.html',
 'toolkit5G-website',
 'toolkit5G',
 'toolkit5G.zip',
 'sample_data']

[ ] import sys
    sys.path.append("/content/toolkit5G-website/api/5G_Toolkit/SymbolMapping")
```

5. Importing SymbolMapping and ChannelCoder directories

```
[ ] from toolkit5G.api.Toolkit import SymbolMapping as sm
    from toolkit5G.api.Toolkit import ChannelCoder as cc
```

6. Defining Autoencoder Parameters

We assume (n,k) = (7,4) here

Parameters of AutoEncoder

```
[ ] ################################################
    # Parameters of a (n,k) AutoEncoder (AE)
    # all the symbols are assumed to be real valued
    ################################################

    # number of information/message symbols that Tx communicates over channel to Rx
    M = 16
    # number of bits per information symbol
    k = int(np.log2(M))
    # number of channel uses or dimension of each code-word symbol or number of bits per code-word symbol
    n = 7
    # Rate of communication. i.e., k bits per n channel uses
    R = k/n
    print("###############################################################################################")
    print("Parameters of "+str((n,k))+" AutoEncoder are:\n")
    print("Number of Information Symbols:" + str(M))
    print("Number of Bits Per Information Symbol:" + str(k))
    print("Number of Channel Uses:" + str(n))
    print("Rate of Communication:" + str(R))

###############################################################################################
```

```
###############################################################
Parameters of (7, 4) AutoEncoder are:

Number of Information Symbols:16
Number of Bits Per Information Symbol:4
Number of Channel Uses:7
Rate of Communication:0.5714285714285714
```

7. SNR to train AutoEncoder

```
[ ]  ##################################
     #SNR at which AE is being trained
     ##################################
     #----------------------------------------------------------------------------------------------------
     # SNR in dB = Es/No, where Es: Energy per symbol, No: Noise Power Spectral Density
     snr_dB = 7
     # snr in linear scale
     snr_train = np.power(10,snr_dB/10)
     # noise standard deviation
     noise_stddev = np.sqrt(1/(2*R*snr_train))
```

8. Training Data

```
[ ]  ############################################
     # One-hot embeddings of information symbols.
     ############################################
     # Each information symbol is mapped to a standard basis vector of dimension M

     symbol_encodings = np.eye(M)
     print("One-Hot Encodings of information symbols:\n")
     print(symbol_encodings)
```

```
One-Hot Encodings of information symbols:

[[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
```

9. Inserting Data Samples

```
[ ]  ##########################################################################
     # Generating data samples of size N. Each sample can take values from 0 to M-1
     ##########################################################################
     N  = 9600000
     #*************************************************************************
     # use this value of N only for Constellation plot when using TSNE under (7,4) AE
     # N = 1500
     #*************************************************************************
     # random indices for labeling information symbols
     train_label = np.random.randint(M,size=N)
     print(train_label)
```

```
[13 10 12 ... 15 13 11]
```

10. Training Data Samples

```
[ ] ########################
    # Training data samples
    ########################
    data = []
    for i in train_label:
        temp = np.zeros(M)
        temp[i] = 1
        data.append(temp)
    # converting data in to a numpy array
    train_data = np.array(data)
    print("\n")
    # Printing the shape of training data
    print("The shape of training data:")
    print(train_data.shape)
```

```
The shape of training data:
(9600000, 16)
```

11. Verify Training Data

```
[ ] # Verifying training data with its label or index for 13 samples

    tempLabel_train = np.random.randint(N,size=13)
    print(tempLabel_train)
    print("\n")
    for i in tempLabel_train:
        print(train_label[i],train_data[i])
```

```
[ 960671 5786607 8601088 4439732 1157617 3487210 8632173 6114654 1332239
 4319694 5144095 3777792 4464441]


14 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
6 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
4 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
3 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
15 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
5 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
5 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
6 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
0 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
0 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
1 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
14 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
9 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
```

12. Generate data for testing Data

```
[ ] ##############################
    # Generating data for testing
    ##############################
    N_test = 16000
    #***********************************************
    # use this only for Constellation Plot of (7,4) AE
    # N_test = 500
    #***********************************************
    test_label = np.random.randint(M,size=N_test)
    test_data = []
    for i in test_label:
        temp    = np.zeros(M)
        temp[i] = 1
        test_data.append(temp)
    # converting it to a numpy array
    test_data = np.array(test_data)
    # Printing the shape of test data
    print("The shape of test data is:")
    print(test_data.shape)
```

```
The shape of test data is:
(16000, 16)
```

13. Verifying Test Data

```
[ ] # Verifying test data with its label for 7 sample
    tempTestLabel = np.random.randint(N_test,size=7)
    print(tempTestLabel)
    print("\n")
    for i in tempTestLabel:
        print(test_label[i],test_data[i])
```

```
[ 1544  8396 10436 10789  7168  3871  9346]


5 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
3 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
2 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
10 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
0 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
15 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
15 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
```

14. Declaring Normalization Functions

```
[ ] from tensorflow.keras import backend as K

    def normalizeAvgPower(x):
        """ Normalizes the power of input tensor"""
        return x / K.sqrt(K.mean(K.square(x)))
```

```
[ ] def normalizeEnergy(x):
        """ Normalizes the energy of input tensor"""
        return np.sqrt(n)*(K.l2_normalize(x,axis=-1))
```

15. Defining Encoder Model

```
[ ] # Define the model
    onehot = Input(shape=(M,))
    dense1 = Dense(M, activation='relu')(onehot)
    dense2 = Dense(n, activation='linear')(dense1)

    # Specify output shape for Lambda layer
    x = Lambda(normalizeAvgPower, output_shape=(n,))(dense2)  # Avg power constraint

    # Channel layer
    y = GaussianNoise(stddev=noise_stddev)(x)

    # Rx layer
    dense3 = Dense(M, activation='relu')(y)
    prob = Dense(M, activation='softmax')(dense3)

    # Define end-to-end autoencoder model
    autoEncoder = Model(onehot, prob)

    # Instantiate optimizer
    adam = Adam(learning_rate=0.01)

    # Compile end-to-end model
    autoEncoder.compile(optimizer=adam, loss='categorical_crossentropy')

    # Print summary of layers and its trainable parameters
    print(autoEncoder.summary())
```

```
Model: "functional_7"
```

Layer (type)	Output Shape	Param #
input_layer_9 (InputLayer)	(None, 16)	0
dense_32 (Dense)	(None, 16)	272
dense_33 (Dense)	(None, 7)	119
lambda_9 (Lambda)	(None, 7)	0
gaussian_noise_7 (GaussianNoise)	(None, 7)	0
dense_34 (Dense)	(None, 16)	128
dense_35 (Dense)	(None, 16)	272

```
Total params: 791 (3.09 KB)
Trainable params: 791 (3.09 KB)
Non-trainable params: 0 (0.00 B)
None
```

16. Training the Autoencoder

In the following code snippet it has been shown how to train an end to end AE by a call to **fit()** method specifying the training and validation data. 50 epochs have been chosen with a batch_size of 1024. One can vary these values to obtain a different trainable model.

```
[ ] ########################
    # Training Auto Encoder
    ########################
    autoEncoder.fit(train_data, train_data,
                    epochs = 50,
                    batch_size = 8*1024,
                    validation_data=(test_data, test_data))
```

Output:

Epoch 1/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **16s** 11ms/step - loss: 0.2284 - val_loss: 1.4727e-06

Epoch 2/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **13s** 11ms/step - loss: 0.0014 - val_loss: 5.1163e-08

Epoch 3/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **13s** 10ms/step - loss: 9.3996e-04 - val_loss: 0.0000e+00

Epoch 4/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **21s** 11ms/step - loss: 7.8140e-04 - val_loss: 0.0000e+00

Epoch 5/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **21s** 11ms/step - loss: 7.1724e-04 - val_loss: 0.0000e+00

Epoch 6/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 6.6487e-04 - val_loss: 0.0000e+00

Epoch 7/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 6.4245e-04 - val_loss: 0.0000e+00

Epoch 8/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **13s** 11ms/step - loss: 6.1045e-04 - val_loss: 0.0000e+00

Epoch 9/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 5.5159e-04 - val_loss: 0.0000e+00

Epoch 10/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **21s** 11ms/step - loss: 5.2714e-04 - val_loss: 0.0000e+00

Epoch 11/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 5.3954e-04 - val_loss: 0.0000e+00

Epoch 12/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **13s** 11ms/step - loss: 5.3096e-04 - val_loss: 0.0000e+00

Epoch 13/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.9291e-04 - val_loss: 0.0000e+00

Epoch 14/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 5.2448e-04 - val_loss: 0.0000e+00

Epoch 15/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **21s** 11ms/step - loss: 5.3740e-04 - val_loss: 0.0000e+00

Epoch 16/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.9086e-04 - val_loss: 0.0000e+00

Epoch 17/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 5.1491e-04 - val_loss: 0.0000e+00

Epoch 18/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 5.0862e-04 - val_loss: 0.0000e+00

Epoch 19/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **13s** 11ms/step - loss: 4.6262e-04 - val_loss: 0.0000e+00

Epoch 20/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.9762e-04 - val_loss: 0.0000e+00

Epoch 21/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **13s** 10ms/step - loss: 4.7144e-04 - val_loss: 0.0000e+00

Epoch 22/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **21s** 11ms/step - loss: 4.6511e-04 - val_loss: 0.0000e+00

Epoch 23/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.9596e-04 - val_loss: 0.0000e+00

Epoch 24/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.4538e-04 - val_loss: 0.0000e+00

Epoch 25/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.9119e-04 - val_loss: 0.0000e+00

Epoch 26/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.9807e-04 - val_loss: 0.0000e+00

Epoch 27/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.6350e-04 - val_loss: 0.0000e+00

Epoch 28/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.6979e-04 - val_loss: 0.0000e+00

Epoch 29/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **23s** 13ms/step - loss: 4.7938e-04 - val_loss: 0.0000e+00

Epoch 30/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **18s** 11ms/step - loss: 4.6818e-04 - val_loss: 0.0000e+00

Epoch 31/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.5606e-04 - val_loss: 0.0000e+00

Epoch 32/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **21s** 11ms/step - loss: 4.4972e-04 - val_loss: 0.0000e+00

Epoch 33/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.2688e-04 - val_loss: 0.0000e+00

Epoch 34/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **24s** 13ms/step - loss: 4.2801e-04 - val_loss: 0.0000e+00

Epoch 35/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.5275e-04 - val_loss: 0.0000e+00

Epoch 36/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.5428e-04 - val_loss: 0.0000e+00

Epoch 37/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.6612e-04 - val_loss: 0.0000e+00

Epoch 38/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.5398e-04 - val_loss: 0.0000e+00

Epoch 39/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.3694e-04 - val_loss: 0.0000e+00

Epoch 40/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.2223e-04 - val_loss: 0.0000e+00

Epoch 41/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **21s** 11ms/step - loss: 4.7823e-04 - val_loss: 0.0000e+00

Epoch 42/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.4710e-04 - val_loss: 0.0000e+00

Epoch 43/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.2942e-04 - val_loss: 0.0000e+00

Epoch 44/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **21s** 11ms/step - loss: 4.6584e-04 - val_loss: 0.0000e+00

Epoch 45/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.6519e-04 - val_loss: 0.0000e+00

Epoch 46/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **13s** 11ms/step - loss: 4.4159e-04 - val_loss: 0.0000e+00

Epoch 47/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.1887e-04 - val_loss: 0.0000e+00

Epoch 48/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **13s** 11ms/step - loss: 4.6371e-04 - val_loss: 0.0000e+00

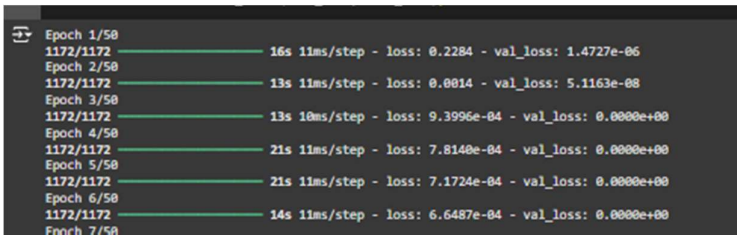
Epoch 49/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **14s** 11ms/step - loss: 4.2782e-04 - val_loss: 0.0000e+00

Epoch 50/50

1172/1172 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ **20s** 11ms/step - loss: 4.2577e-04 - val_loss: 0.0000e+00

<keras.src.callbacks.history.History at 0x7dc6c2f36680>



17. Defining Transmitter (Tx) and Receiver (Rx) channel from Trained AutoEncoder

```
#########################################################
# Defining Tx from end to end trained autoEncoder model
#########################################################

transmitter   = Model(onehot, x)
#*******************************************************
######################
# Defining channel part
######################
channelInput  = Input(shape=(n,))
channelOutput = autoEncoder.layers[-3](channelInput)
channel       = Model(channelInput, channelOutput)
#*******************************************************
#################
# Defining Rx part
#################
rxInput       = Input(shape=(n,))
rx1           = autoEncoder.layers[-2](rxInput)
rxOutput      = autoEncoder.layers[-1](rx1)
receiver      = Model(rxInput,rxOutput)
```

18. Analyzing BER Performance

The following code snippet computes and plots BLER performance of of (n,k) AE and compares it with base line (n,k) Hamming code.

```
[ ]  ####################################
     # SNR vs BLER computation and plotting
     ####################################
     # use this snr_dB = np.arange(0,15,2) for (n,k) = (2,4) or (2,2) AE
     # use this snr_dB = np.arange(-4,8.5,0.5) for (7,4) AE

     snr_dB = np.arange(-4,8.5,0.5)
     bler   = np.zeros(snr_dB.shape[0])
     for ii in range(0,snr_dB.shape[0]):
         snr_linear      = 10.0**(snr_dB[ii]/10.0)      # snr in linear scale
         noise_std       = np.sqrt(1/(2*R*snr_linear))
         noise_mean      = 0
         num_errors      = 0
         num_samples     = N_test
         #------------------------------------------------------------
         noise           = noise_std*np.random.randn(num_samples,n)
         #------------------------------------------------------------
         ########################
         # predicted input symbols
         ########################
         x_hat           = transmitter.predict(test_data)
         #------------------------------------------------------------
         #############
         # noisy input
         #############
         x_hat_noisy     = x_hat + noise
         #------------------------------------------------------------
         ########################
         # predicted output symbols
         ########################
         y_hat           = receiver.predict(x_hat_noisy)
         #------------------------------------------------------------
         #################
         # symbol estimates
         #################
         sym_estimates   = np.argmax(y_hat, axis=1)
         #------------------------------------------------------------
         ############################################
         # counting errors and computing bler at each snr
         ############################################
         num_errors      = int(np.sum(sym_estimates != test_label))
         bler[ii]        = num_errors/num_samples
         print('SNR(dB):', snr_dB[ii], 'BLER:', bler[ii])
```

Output:

500/500 ———————————————————— **1s** 1ms/step

500/500 ———————————————————— **1s** 1ms/step

SNR(dB): -4.0 BLER: 0.5031875

500/500 ———————————————————— **1s** 1ms/step

500/500 ———————————————————— **1s** 2ms/step

SNR(dB): -3.5 BLER: 0.4609375

500/500 ━━ **1s** 2ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): -3.0 BLER: 0.41725

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): -2.5 BLER: 0.38925

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): -2.0 BLER: 0.3445625

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): -1.5 BLER: 0.300125

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): -1.0 BLER: 0.263

500/500 ━━ **1s** 2ms/step

500/500 ━━ **1s** 2ms/step

SNR(dB): -0.5 BLER: 0.2259375

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 0.0 BLER: 0.1869375

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 0.5 BLER: 0.153625

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 1.0 BLER: 0.121125

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 1.5 BLER: 0.093125

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 2.0 BLER: 0.0665

500/500 ━━ **1s** 2ms/step

500/500 ━━ **1s** 2ms/step

SNR(dB): 2.5 BLER: 0.0486875

500/500 ━━ **1s** 2ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 3.0 BLER: 0.0338125

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 3.5 BLER: 0.021125

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 4.0 BLER: 0.0130625

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 4.5 BLER: 0.0074375

500/500 ━━ **1s** 1ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 5.0 BLER: 0.004125

500/500 ━━ **1s** 2ms/step

500/500 ━━ **1s** 2ms/step

SNR(dB): 5.5 BLER: 0.00175

500/500 ━━ **1s** 2ms/step

500/500 ━━ **1s** 1ms/step

SNR(dB): 6.0 BLER: 0.001

500/500 ——————————————————————————— **1s** 1ms/step

500/500 ——————————————————————————— **1s** 1ms/step

SNR(dB): 6.5 BLER: 0.0001875

500/500 ——————————————————————————— **1s** 1ms/step

500/500 ——————————————————————————— **1s** 1ms/step

SNR(dB): 7.0 BLER: 0.0001875

500/500 ——————————————————————————— **1s** 1ms/step

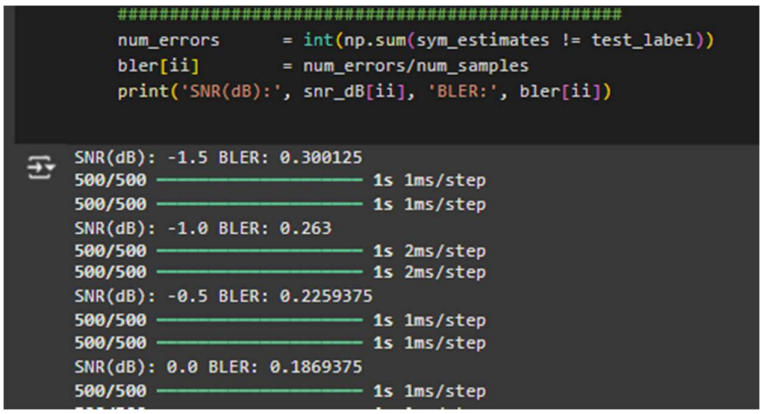
500/500 ——————————————————————————— **1s** 1ms/step

SNR(dB): 7.5 BLER: 0.000125

500/500 ——————————————————————————— **1s** 1ms/step

500/500 ——————————————————————————— **1s** 1ms/step

SNR(dB): 8.0 BLER: 0.0



```
        ###############################################
        num_errors      = int(np.sum(sym_estimates != test_label))
        bler[ii]        = num_errors/num_samples
        print('SNR(dB):', snr_dB[ii], 'BLER:', bler[ii])
```

```
SNR(dB): -1.5 BLER: 0.300125
500/500 ——————————————————— 1s 1ms/step
500/500 ——————————————————— 1s 1ms/step
SNR(dB): -1.0 BLER: 0.263
500/500 ——————————————————— 1s 2ms/step
500/500 ——————————————————— 1s 2ms/step
SNR(dB): -0.5 BLER: 0.2259375
500/500 ——————————————————— 1s 1ms/step
500/500 ——————————————————— 1s 1ms/step
SNR(dB): 0.0 BLER: 0.1869375
500/500 ——————————————————— 1s 1ms/step
```

19. Hamming Code Transmitter

```python
import numpy as np

class HammingEncoder:
    def __init__(self, k, n):
        self.k = k  # Message length
        self.n = n  # Codeword length

    def encode(self, bits):
        if bits.shape[1] != self.k:
            raise ValueError(f"Input bits should have {self.k} columns.")

        # Calculate the parity bits for Hamming code
        m = self.n - self.k
        encoded_bits = np.zeros((bits.shape[0], self.n), dtype=int)

        for i in range(bits.shape[0]):
            message = bits[i]
            parity = np.zeros(m, dtype=int)

            # Set up message bits in their positions in the encoded word
            j = 0
            for pos in range(1, self.n + 1):
                if not (pos & (pos - 1)) == 0:  # Non-power of 2 positions for message bits
                    encoded_bits[i, pos - 1] = message[j]
                    j += 1

            # Calculate parity bits based on message bits
            for p in range(m):
                parity_bit_position = 2 ** p
                parity[p] = np.sum(encoded_bits[i, np.where(np.bitwise_and(np.arange(1, self.n + 1), parity_bit_position) != 0)]) % 2
                encoded_bits[i, parity_bit_position - 1] = parity[p]

        return encoded_bits

    def __call__(self, bits):
        return self.encode(bits)
```

20. Defining Mapper Class for modulation Scheme

```python
class Mapper:
    def __init__(self, constellation_type, num_bits_per_symbol):
        self.constellation_type = constellation_type
        self.num_bits_per_symbol = num_bits_per_symbol
        self.symbols = self.generate_constellation()

    def generate_constellation(self):
        if self.constellation_type == "bpsk":
            return np.array([-1, 1])  # BPSK constellation points
        elif self.constellation_type == "qpsk":
            return np.array([-1-1j, -1+1j, 1-1j, 1+1j])  # QPSK constellation points
        else:
            raise ValueError("Unsupported constellation type")

    def map(self, bits):
        if len(bits[0]) % self.num_bits_per_symbol != 0:
            raise ValueError("Number of bits must be a multiple of num_bits_per_symbol")

        symbols = []
        for i in range(bits.shape[0]):
            for j in range(0, len(bits[i]), self.num_bits_per_symbol):
                bit_group = bits[i][j:j + self.num_bits_per_symbol]
                index = int("".join(map(str, bit_group)), 2)
                symbols.append(self.symbols[index])

        return np.array(symbols)

    def __call__(self, bits):
        return self.map(bits)
```

21. Defining inputs

```python
# Hamming Code Configurations
m = 3
k = 2**m - m - 1
n = 2**m - 1

# Payload Generation
numDim = 2
n1     = 1000000
bits   = np.random.randint(2, size=(n1, k))

# Hamming Encoder
hamming_encoder = HammingEncoder(k, n)
encBits = hamming_encoder(bits)

# Rate Matching
codeword = encBits

# Symbol Mapping
constellation_type  = "bpsk"
num_bits_per_symbol = 1

mapperObject = Mapper(constellation_type, num_bits_per_symbol)
symbols = mapperObject(codeword)

print()
print("******** (" + str(n) + "," + str(k) + ") Hamming Code ********")
print("       Shape of Input: " + str(bits.shape))
print("    Shape of Enc Bits: " + str(encBits.shape))
print("   Constellation type: " + str(constellation_type))
print("Number of bits/symbol: " + str(num_bits_per_symbol))
print("*******************************")
print()
```

```
******** (7,4) Hamming Code ********
       Shape of Input: (1000000, 4)
    Shape of Enc Bits: (1000000, 7)
   Constellation type: bpsk
Number of bits/symbol: 1
*******************************
```

22. Defining Hamming Decoder

```
[ ]  import numpy as np

     class HammingDecoder:
         def __init__(self, k, n):
             self.k = k  # Number of data bits
             self.n = n  # Number of coded bits
             self.parity_matrix, self.generator_matrix = self._generate_hamming_matrices()

         def _generate_hamming_matrices(self):
             # Parity-check matrix for (7,4) Hamming code
             H = np.array([[1, 1, 1, 0, 1, 0, 0],
                           [1, 1, 0, 1, 0, 1, 0],
                           [1, 0, 1, 1, 0, 0, 1]], dtype=int)
             # Generator matrix for (7,4) Hamming code
             G = np.array([[1, 0, 0, 0, 1, 1, 1],
                           [0, 1, 0, 0, 1, 1, 0],
                           [0, 0, 1, 0, 1, 0, 1],
                           [0, 0, 0, 1, 0, 1, 1]], dtype=int)
             return H, G

         def decode_hard(self, received):
             # Hard decision decoding with error correction
             decoded = []
             for r in received:
                 syndrome = np.mod(np.dot(self.parity_matrix, r.T), 2)
                 syndrome_idx = int("".join(map(str, syndrome)), 2)
                 if 0 < syndrome_idx <= self.n:
                     r[syndrome_idx - 1] ^= 1  # Correct the error
                 decoded.append(r[:self.k])  # Extract original data bits
             return np.array(decoded)

         def decode_soft(self, llrs, decoding_type="sphereDecoding"):
             # Soft decision decoding based on LLRs
             if decoding_type == "sphereDecoding":
                 return self.decode_hard(np.where(llrs > 0, 1, 0))
             else:
                 raise ValueError("Unsupported soft decoding method")

         def __call__(self, data, decoding_type="hard"):
             if decoding_type == "hard":
                 return self.decode_hard(data)
             elif decoding_type == "sphereDecoding":
                 return self.decode_soft(data, decoding_type)
```

23. Defining Demapper

```python
class Demapper:
    def __init__(self, method, constellation_type, num_bits_per_symbol, hard_out=False):
        self.method = method
        self.constellation_type = constellation_type
        self.num_bits_per_symbol = num_bits_per_symbol
        self.hard_out = hard_out
        self.symbol_map = self.generate_constellation()

    def generate_constellation(self):
        if self.constellation_type == "bpsk":
            return np.array([-1, 1])  # BPSK
        else:
            raise ValueError("Unsupported constellation type")

    def demap(self, symbols, snr_inv):
        llrs = []
        for symbol in symbols:
            if self.constellation_type == "bpsk":
                if self.method == "app":
                    llr = 2 * symbol.real / snr_inv  # LLR computation for BPSK
                elif self.method == "maxlog":
                    llr = np.sign(symbol.real)  # Simplified for max-log method
                llrs.append(llr)
        llrs = np.array(llrs)

        if self.hard_out:
            return np.where(llrs > 0, 1, 0)  # Hard output
        return llrs  # Soft output

    def __call__(self, data):
        return self.demap(*data)
```

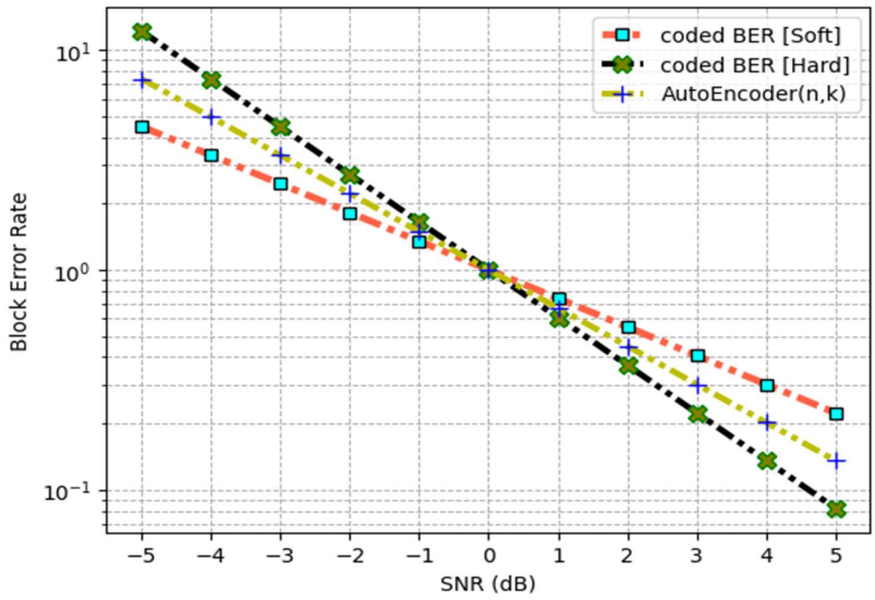
24. Building the system

```python
# Simulation Parameters
snr_dB = np.array([-5,-4,-3,-2,-1,0,1,2,3,4,5])  # Example SNR values in dB
SNR = 10 ** (snr_dB / 10)      # SNR in linear scale
codedBLERhard = np.zeros(SNR.shape)
codedBLERsoft = np.zeros(SNR.shape)

# Symbol Demapping Configuration
demapping_method = "app"
hard_out = False
constellation_type = "bpsk"
num_bits_per_symbol = 1

# Initialize Demapper
demapper = Demapper(demapping_method, constellation_type, num_bits_per_symbol, hard_out=hard_out)

# Encoder and Decoder Setup
m = 3
k = 2**m - m - 1
n = 2**m - 1
hamming_decoder = HammingDecoder(k, n)

# Generate random bits and encode
num_bits = 1000
bits = np.random.randint(0, 2, size=(num_bits, k))
encBits = np.random.randint(0, 2, size=(num_bits, n))  # Dummy encoded bits
codeword = encBits

# Map the encoded bits to symbols (BPSK)
symbols = 2 * codeword - 1  # BPSK: Map 0 -> -1, 1 -> 1

# BLER Simulation Loop
for snrIndex, snr in enumerate(SNR):
    # Add noise to the symbols
    noise = np.sqrt(0.5 / snr) * (np.random.standard_normal(symbols.shape) + 1j * np.random.standard_normal(symbols.shape))
    noisy_symbols = symbols + noise

    # Soft and Hard Demapping
    llrEst = demapper([noisy_symbols, np.float32(1 / snr)])
    uncBits = np.where(llrEst > 0, np.int8(1), np.int8(0))

    # Hard Decoding
    decBits_hard = hamming_decoder(uncBits, "hard")
    codedBLERhard[snrIndex] = np.mean(np.where(np.sum(np.abs(bits - decBits_hard), axis=1) > 0, True, False))

    # Soft Decoding
    decBits_soft = hamming_decoder(llrEst, "sphereDecoding")
    codedBLERsoft[snrIndex] = np.mean(np.where(np.sum(np.abs(bits - decBits_soft), axis=1) > 0, True, False))

    # Print results for each SNR
    print(f"At SNR(dB): {snr_dB[snrIndex]} | coded BLER (soft): {codedBLERsoft[snrIndex]:.5f} | coded BLER (hard): {codedBLERhard[snrIndex]:.5f}")
    snrIndex += 1
```

```
At SNR(dB): -5 | coded BLER (soft): 0.94300 | coded BLER (hard): 0.94300
At SNR(dB): -4 | coded BLER (soft): 0.93500 | coded BLER (hard): 0.93500
At SNR(dB): -3 | coded BLER (soft): 0.95600 | coded BLER (hard): 0.95600
At SNR(dB): -2 | coded BLER (soft): 0.93800 | coded BLER (hard): 0.93800
At SNR(dB): -1 | coded BLER (soft): 0.94400 | coded BLER (hard): 0.94400
At SNR(dB): 0 | coded BLER (soft): 0.94200 | coded BLER (hard): 0.94200
At SNR(dB): 1 | coded BLER (soft): 0.95300 | coded BLER (hard): 0.95300
At SNR(dB): 2 | coded BLER (soft): 0.94600 | coded BLER (hard): 0.94600
At SNR(dB): 3 | coded BLER (soft): 0.94600 | coded BLER (hard): 0.94600
At SNR(dB): 4 | coded BLER (soft): 0.94900 | coded BLER (hard): 0.94900
At SNR(dB): 5 | coded BLER (soft): 0.94700 | coded BLER (hard): 0.94700
```

25. BLER plot : comparison of AutoEncoder BLER with base line (n,k) Hamming Code BLER

```
[ ]  import matplotlib.pyplot as plt
     import numpy as np

     # Example Data: Replace with actual data if available
     snr_dB = np.array([-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5])  # SNR values from -5 to 5
     codedBLERsoft = np.exp(-0.3 * snr_dB)  # Example data for coded BLER [Soft]
     codedBLERhard = np.exp(-0.5 * snr_dB)  # Example data for coded BLER [Hard]
     bler = np.exp(-0.4 * snr_dB)  # Example data for AutoEncoder BLER

     # Create figure and axis
     fig, ax = plt.subplots()

     # Plotting coded BLER [Soft]
     ax.semilogy(
         snr_dB, codedBLERsoft, 'tomato', lw=3.5,
         linestyle=(0, (3, 1, 1, 1, 1, 1)), marker="s", ms=6,
         mec="k", mfc="cyan", label="coded BER [Soft]"
     )

     # Plotting coded BLER [Hard]
     ax.semilogy(
         snr_dB, codedBLERhard, 'k', lw=3,
         linestyle=(0, (3, 1, 1, 1, 1, 1)), marker="X", ms=9,
         mec="green", mfc="olive", label="coded BER [Hard]"
     )

     # Plotting AutoEncoder(n,k) BLER
     ax.semilogy(
         snr_dB, bler, 'y', lw=3,
         linestyle=(0, (3, 1, 1, 1, 1, 1)), marker="+", ms=9,
         mec="blue", mfc="pink", label="AutoEncoder(n,k)"
     )

     # Adding legend, labels, and grid
     ax.legend()
     ax.set_xticks(snr_dB)  # Set x-ticks to match the SNR values
     ax.grid(True, which='both', linestyle='--', linewidth=0.7)  # Grid for better visualization
     ax.set_ylabel("Block Error Rate")
     ax.set_xlabel("SNR (dB)")

     # Display the plot
     plt.show()
```



27.Constellation Learning

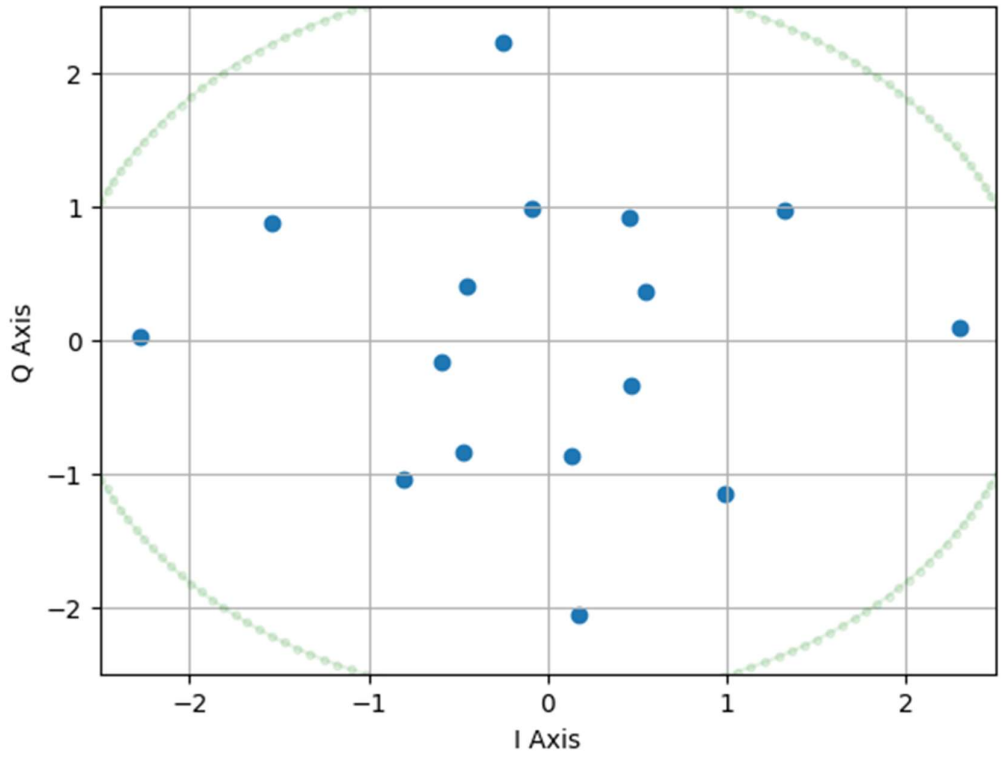
The following cells provide the code to generate and plot the learned constellation by Tx in the presence of AWGN.

For (2,4) and (2,2) we can use same N,N_test for BLER and constellation plots.

But (7,4) will use TSNE

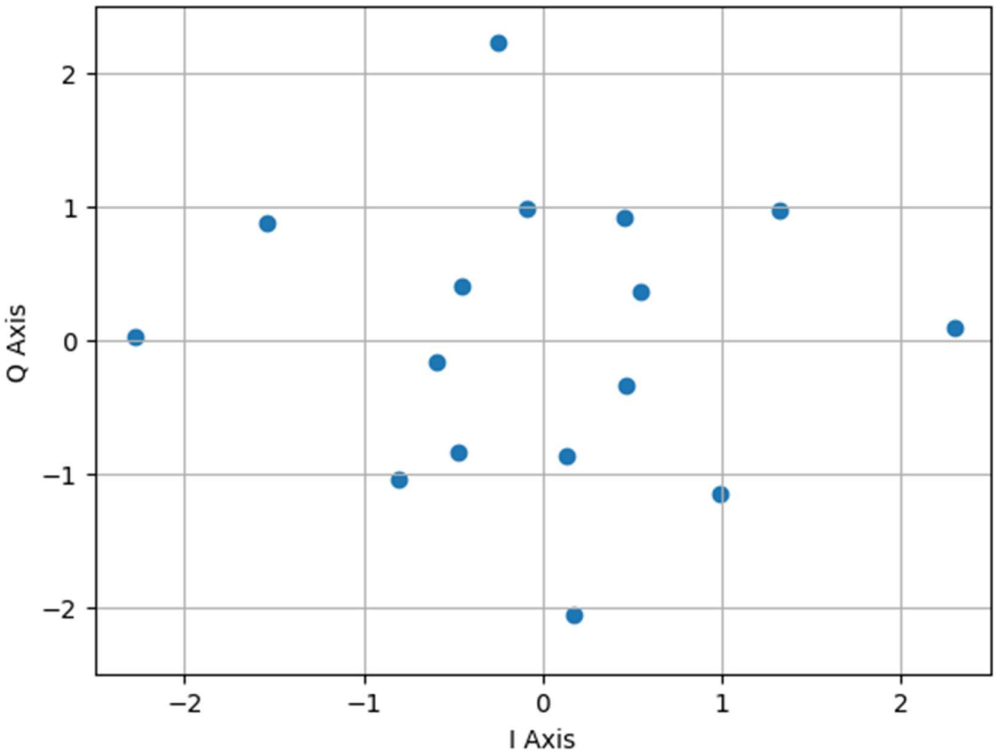
constellation under energy constraint for (n,k) = (2,4) or (2,2)

```
[ ]  ################################################################################
     # plotting learned constellation under energy constraint for (n,k) = (2,4) or (2,2)
     ################################################################################
     constellationPoints = transmitter.predict(symbol_encodings)
     r = np.linalg.norm(constellationPoints[0])
     theta = np.linspace(0,2*np.pi,200)
     fig, ax = plt.subplots()
     ax.scatter(constellationPoints[:,0],constellationPoints[:,1])
     ax.plot(r*np.cos(theta),r*np.sin(theta), c="green", alpha = 0.1, marker=".")
     plt.axis((-2.5,2.5,-2.5,2.5))
     plt.grid()
     plt.xlabel('I Axis')
     plt.ylabel('Q Axis')
     plt.show()
```



Constellation under average power constraint for (n,k) = (2,4) or (2,2)

```
##############################################################################
# plotting learned constellation under average power constraint for (n,k) = (2,4) or (2,2)
##############################################################################

fig, ax = plt.subplots()
ax.scatter(constellationPoints[:,0],constellationPoints[:,1])
plt.axis((-2.5,2.5,-2.5,2.5))
plt.grid()
plt.xlabel('I Axis')
plt.ylabel('Q Axis')
plt.show()
```


Q Axis
I Axis
2
1
0
−1
−2
−2
−1
0
1
2

6. Applications of Autoencoder

Image and Audio Compression: Autoencoders can compress huge images or audio files while maintaining most of the vital information. An autoencoder is trained to recover the original picture or audio file from a compressed representation.

Anomaly Detection: One can detect anomalies or outliers in datasets using autoencoders. Training the autoencoder on a dataset of normal data and any input that the autoencoder cannot accurately reconstruct is called an anomaly.

Dimensionality Reduction: Autoencoders can lower the dimensionality of high-dimensional datasets. We can accomplish this by teaching an autoencoder a lower-dimensional data representation that captures the most relevant features.

Data Generation: Employ autoencoders to generate new data similar to the training data. One can accomplish this by sampling from the autoencoder's compressed representation and then utilizing the decoder to create new data.

Denoising: One can utilize autoencoders to reduce noise from data. We can accomplish this by teaching an autoencoder to recover the original data from a noisy version.

Recommender System: Using autoencoders, we can use users' preferences to generate personalized suggestions. We can accomplish this by training an autoencoder to learn a compressed representation of the user's history of system interactions and then utilizing this representation to forecast the user's preferences for new items.

7. Conclusion and Future Work

Conclusion

In this study, we implemented and analyzed an end-to-end digital communication system modeled as an Auto Encoder. The Auto Encoder approach demonstrated the ability to jointly optimize the transmitter (Tx) and receiver (Rx) for communication over an Additive White Gaussian Noise (AWGN) channel. Key findings include:

1. **Performance Comparison**:
   The Auto Encoder-based system was compared with traditional communication systems utilizing (n, k) Hamming Codes. While the Hamming Code provides reliable error correction for specific block lengths and fixed alphabet, the Auto Encoder autonomously learns a signaling scheme tailored to the channel conditions, often resulting in performance gains, especially at higher noise levels (lower SNRs).

2. **Constellation Learning**:
   The Auto Encoder successfully learned robust signaling constellations that adapt to the noise characteristics of the channel, outperforming traditional fixed constellations such as BPSK and QPSK in some scenarios.

3. **Joint Optimization**:
   Unlike traditional systems where Tx and Rx are designed separately, the Auto Encoder jointly optimized the communication pipeline, leading to better resilience to noise.

4. **Flexibility and Adaptability**:
   The Auto Encoder's learned scheme was adaptable to various channel conditions without requiring explicit knowledge of the channel model, which is often a limitation in traditional systems.

Future Work

1. **Extension to Other Channel Models**:
   Investigate the performance of Auto Encoder-based communication systems under different channel models, such as Rayleigh fading, Rician channels, and more complex environments like MIMO systems.

2. **Higher-Dimensional Constellations**:
   Explore the performance of Auto Encoders for higher-order constellations (e.g., 16-QAM, 64-QAM) to evaluate their efficiency in high-data-rate scenarios.

3. **Incorporating Channel Coding**:
   Integrate traditional channel coding techniques (e.g., Turbo Codes, LDPC) with the Auto Encoder to assess combined performance improvements.

4. **Robustness to Channel Variability**:
   Train Auto Encoders on dynamically varying channel conditions to develop a robust communication system capable of real-time adaptation.

5. **Hardware Implementation**:
   Implement and test the Auto Encoder-based communication system on hardware (e.g., Software Defined Radios) to evaluate its practicality and performance in real-world scenarios.

6. **Comparison with Advanced Codes**:
   Extend the comparison to advanced coding techniques like Polar Codes and Neural Decoders to benchmark Auto Encoder performance against state-of-the-art methods.

7. **Low-Latency and Energy-Efficient Designs**:
   Focus on designing Auto Encoders optimized for low-latency and energy efficiency, which are critical for IoT and 5G/6G applications.

Reference: [1] T. O'Shea and J. Hoydis, "An Introduction to Deep Learning for the Physical Layer," in IEEE Transactions on Cognitive Communications and Networking, vol. 3, no. 4, pp. 563-575, Dec. 2017, doi: 10.1109/TCCN.2017.2758370.