Contents

Abstract ....

Introduction 1

Objectives: 2

Defining an Autoencoder 2

Understanding the Problem 2

AI & Information Theory in practice 3

Data Compression 3

Hamming code 3

Designing the Autoencoder 4

A Theoretical Approach 4

Practical Implementation (with code snippets) 4

Building the transmitter: 5

Building the Channel: 6

Building the Receiver: 6

Defining the Loss Function: 6

Optimizing the 'Autoencoding' Algorithm: 7

Defining a SNR Function: 7

Training the Autoencoder: 7

Evaluating the autoencoder : 8

Channel capacity 9

Modulation Schemes vs Autoencoder: 11

Results and Observations : 13

Comparing Losses among different Optimizer Settings: 13

Comparing Autoencoder against Modulation Schemes: 14

Observations: 15

Conclusion: 16

References: 17

Abstract

Throughout the history of mankind, ever since our origin, we as humans have sought after ways to find algorithms to attain efficient transfer of information over a variety of communication channels. Through technological advancements now we have efficient ways to communicate over long distances and hard mediums. These range from thermal noise limited channels exhibiting simple Gaussian noise-like behaviors to much more complex channels exhibiting multi-path fading.

The fundamental aim of '**Information Theory**' is to find a way to establish a reliable communication over an unreliable channel. All in all, we need to send a message through the transmitter which will further travel through the channel whose inherent job is to add some noise to our original message and then at the receiving end we need to find a way to reconstruct the message which closely resembles the original one.

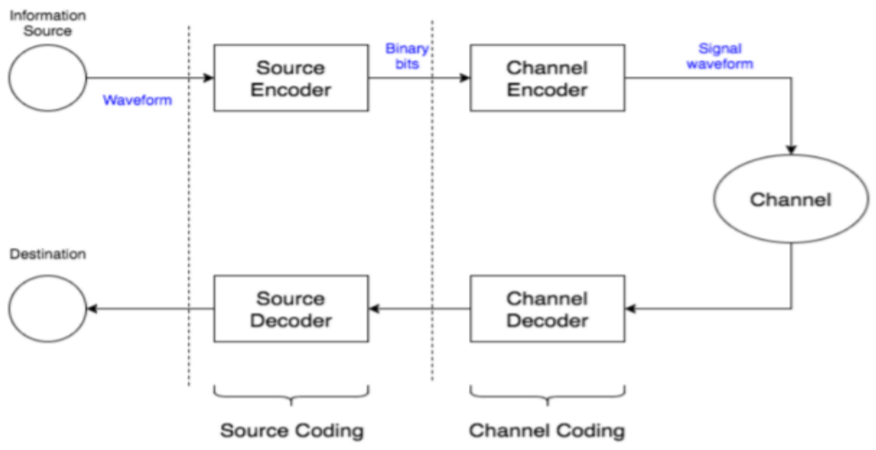
To solve this problem one can choose between two approaches. The **Physical solution** is to get better hardware or get better insulation for the wired network and change some system related physics. The better approach is through a **system solution**. We can add encoding and decoding systems to increase the reliability of our channel. An even better solution today is through Artificial Intelligence. Today, new variations in wireless systems through **artificial intelligence** can learn from the data and optimize their utilization to upgrade their performance.

Through this project we address the problem of learning efficient and adaptive ways to communicate binary information over an impaired channel. Here we train an autoencoder for information transmission over an end-to-end communication system, where the encoder will replace the transmitter tasks such as modulation and coding along with adding redundancy and the decoder will replace the receiver tasks such as demodulation and decoding.

In the end we will compare our Autoencoder’s performance with different Modulation schemes like QPSK and 8PSK.

Introduction

The goal of a communication system is to deliver a certain message (text, audio, image, video, etc.) to the destination over a noisy physical channel (copper wire, optical fiber, EM wave radiation, etc.). Digital communication systems are communication systems that use digital sequence (typically binary digits, that is, bits) as an interface between the source coding part and the channel coding part. Its components can be represented as:



When handling the complexity of optimization for new wireless applications with high degrees of freedom, these systems often fail. The reason being their probabilistic approaches to infer the source message. Deep learning displays a promising potential when it comes to addressing this challenge via data-driven solutions.

Instead of traversing through a rigid pre-planned design, new generations of wireless systems empowered by deep learning can learn from data and optimize their performance. These smart communication systems take aid from several trivial machine learning tasks like detection, classification, and prediction.

Also, Wireless communications data comes at colossal columns along with high rates of transfer. At the same time, it is exposed to several interference and security threats due to the shared nature of the medium. Conventional modeling lacks the capacity to capture the relationship between highly complex data, whereas deep learning comes with the merit to fulfill the requirements while taking care of data rate, latency, efficiency, and traffic capacity.

Through this project we find a way to describe how deep learning models like autoencoders can be used to build a communication system. We will notice how an autoencoder will gradually improve and surpass the performance of conventional systems by optimizing the communication between the transmitter and the receiver together, instead of optimizing their individual modules.

Objectives:

1. To train an Autoencoder for Information Transmission at different hyperparameter tunings and represent it as a standalone end-to-end communication system.

2. To build the parts of the Autoencoder such that they replace the components of a conventional communication system.

3. Performing Hyperparameter tuning to ensure optimal reconstruction of outputs.

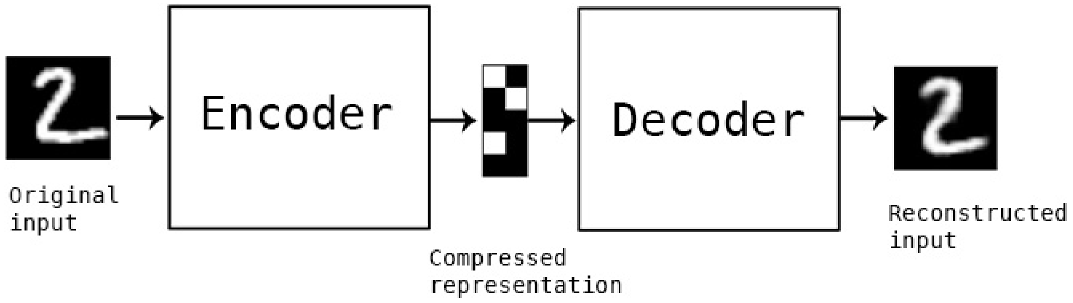
4. Comparing the performance of the autoencoder with different Modulation Schemes.

Defining an Autoencoder

Autoencoding is a data compression algorithm where the compression and decompression functions are data-specific, lossy, and learned automatically from examples rather than engineered by a human.

An autoencoder is a deep neural network that consists of an encoder that learns a (latent) representation of the given data, and a decoder that reconstructs the input data from the encoded data.

One might identify it as unsupervised learning, but in reality, its more inclined towards self-supervised learning. In this Project, modulation and encoding at the transmitter end corresponds to the encoder and decoding with demodulation at the receiver end corresponds to the decoder.



Understanding the Problem

Throughout the years we have achieved numerous discrete ways to communicate over a channel allowing efficient operations at specific SNR (Signal to Noise Ratio) values. Rate matching and code adaptation has allowed us to come up with different ways to establish a reliable communication channel, but many of them are computationally complex in practice and require expensive hardware or DSP software to leverage in mobile radio systems.

By taking the approach of self-supervised learning of an end-to-end communications system by optimization of reconstruction cost in a channel auto-encoder, we seek to learn new methods of modulation which blur the lines between modulation and error correction, providing similar SNR to bit error rate performance (BER) while achieving lower computational complexity requirements at runtime. The potential impact of such a system on the development, deployment and capabilities of wireless systems holds enormous potential.

AI & Information Theory in practice

In this modern ear, almost every field started utilizing features form other areas to improve the efficiency and performance. This project is an example showing that important aspects of the Information Theory such as Hamming code utilizes tools from AI such as autoencoders as a channel of communication.

Data Compression

This is one of the most beneficial areas where autoencoders contribute much to information theory. They outperform PCA due to the fact that autoencoder can learn non-linear transformations with a non-linear activation function and multiple layers, also it can make use of pre-trained layers from another model with much more other advantages.

Autoencoders -if they are properly trained – can outperform traditional image compression standards including JPEG and JPEG2000. So whether if it’s image compression or binary or ternary code compression, autoencoders are always among top options.

One of the applications where we can implement data compression is hamming code.

Hamming code

Hamming code is a set of error-correction codes that can be used to detect and correct the errors that can occur when the data is moved or stored from the sender to the receiver. It is technique developed by R.W. Hamming for error correction.

Hamming codes were one of the earliest block codes used to detect errors in the calculations of the relay-based computers at the time. They are characterized by the structure $(n, k) = (2n − 1, 2\ n − 1 − m)$ where $m = 2, 3, ..$ They can detect up to all combinations of 2 or fewer errors within a block. For example, the (7, 4) binary Hamming Code has $n = 7$, $M = 16$, and dmin = 3.

Encoding Hamming code and decoding it using autoencoders properly would contribute to a better performance and timing along with a decent cost effectiveness. And this is one of the areas where autoencoders and information theory intersect through the transmission of hamming code.

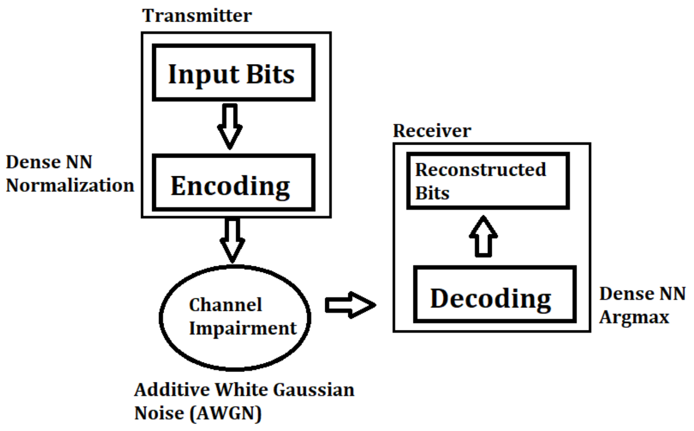
Designing the Autoencoder

A Theoretical Approach

In the case of communication, we consider overcomplete autoencoders. Their purpose is to add (on the transmitter side) and remove (on the receiver side) redundancy to the input message representation in a way that is matched to the channel (noise layer) unlike the undercomplete autoencoders (i.e., whose hidden layers have fewer neurons than the input/output) which have traditionally been studied for extracting hidden features and learning a compressed representation of the input.

The transmitter wants to convey a message “s” over the channel to the receiver. At the receiver side, a noisy and possibly distorted message can be observed. Now, the task of the receiver is to produce the estimate of the original message. The communication system described above can be interpreted as an autoencoder where it is trained to reconstruct the input at the output and, as the information must pass each layer, the network needs to find a robust representation of the input message at every layer.

As opposed to the independent block optimization of conventional communication systems, deep learning is capable to jointly optimize multiple communication blocks at the transmitter and receiver by training them as deep neural networks (DNNs).



Practical Implementation (with code snippets)

We start off by setting up the right environment and making the necessary imports. We need to start with Python (and its numerical libraries such as NumPy. Further, we use Tensorflow along with Keras as our deep learning environment and access it via the Spyder IDE provided with the Anaconda distribution. No GPU is required for training. Our implementation will be handled well by CPU and enough RAM however, the main advantage of having access to a GPU is the support for accelerated training which speeds up the training by (at least) an order of magnitude.

Required Libraries:

1. Tensorflow, to perform computing graph-based training of the NN.
2. Numpy for basic computations and to feed the NN.
3. Matplotlib, to plot our results.

We define the main parameters of the autoencoder, k (Number of information bits per message), n (Number of real channel uses per message) , M (Number of messages = 2**k)

We begin with defining the batch size. The batch size is a hyperparameter of gradient descent that controls the number of training samples to work through before the model's internal parameters are updated. We want this to be a flexible parameter to allow adjustments during training.

```
# AUTOENCODER SYSTEM MODEL
batch_size = tf.placeholder(tf.int32,shape=[])
s = tf.random.uniform(shape=[batch_size],minval=0,maxval=M,dtype=tf.int32)
s_one_hot = tf.one_hot(s,depth=M)
```

The above code displays the process of message creation that we want to transmit. "s" can only take legal combinations of values with a single high '1' bit and all the others low '0'. This encoding allows a state machine to run at a faster clock rate than any other encoding of that state machine. Determining the state of a one-hot vector has a low and constant cost of accessing one flip-flop. They are simply drawn from a random uniform distribution. To feed them to the first dense NN layer of the transmitter part, we transform them to the one-hot vectors. It now holds the batch size vectors of length M, where only one entry is set to 1.0 while all other entries are 0.0.

Building the transmitter:

```
# Transmitter
tx = tf.keras.layers.Dense(units=M,activation="relu")(s_one_hot)
tx = tf.keras.layers.Dense(units=n,activation=None)(tx)
x = tx / tf.sqrt(tf.reduce_mean(tf.square(tx)))
```

For understanding the transmitter part we only need two dense layers to perform a transformation from messages to real valued channel uses. We use Tensorflow/Keras dense layers in the following way. The first dense transmitter layer is "relu" activated and has M neurons because the samples of input "s_one_hot" are also of length M. The second dense layer has n units, which forms the output of the transmitter, and does not have any activation function, since the transmitter outputs real-valued numbers. To prevent the transmitter from learning unnecessarily large outputs and becoming numerically unstable, we normalize the average power of all transmitter outputs in the mini-batch to equal 1.0.

Building the Channel:

Now x is the output of our transmitter, which will be passed on through the channel. We choose a basic additive white Gaussian noise (AWGN) channel that simply adds scaled normal distributed (real-valued) on top of x. But to be able to change the noise power and, thereby, the signal to noise ratio (SNR), we implement it as a feedable Tensorflow placeholder. Then we draw a noise tensor (i.e., a vector per sample in the mini batch) of the same shape as x from a normal distribution with the standard deviation given by the placeholder.

```
# Channel
noise_std = tf.placeholder(dtype=tf.float32,shape=[])
noise = tf.random.normal(shape=tf.shape(x),stddev=noise_std)
y = x + noise
```

Now, we simply add this random noise tensor on top of x to get y, which is the received message after transmission.

Building the Receiver:

Since the channel now acts as the penalty layer for our autoencoder, we need a receiver part that produces a reproduction of our original message “s_hat:” given y. This receiver part consists of a first dense layer with M neurons and “relu” activation. Receiver is described by one input and one output layer. The output layer has M units, to produce an estimate on the probability of each possible message wherein no activation function is used.

```
# Receiver
rx = tf.keras.layers.Dense(units=M, activation="relu")(y)
s_hat = tf.keras.layers.Dense(units=M, activation=None)(rx)
```

Defining the Loss Function:

After building our Autoencoder, we can feed in messages and get the corresponding reconstructed messages as an output. At this stage we defined a loss function that calculates the current performance of the model by comparing the input s with the output s_hat. We use a cross entropy loss function that activates the output with “softmax” and accepts sparse labels.

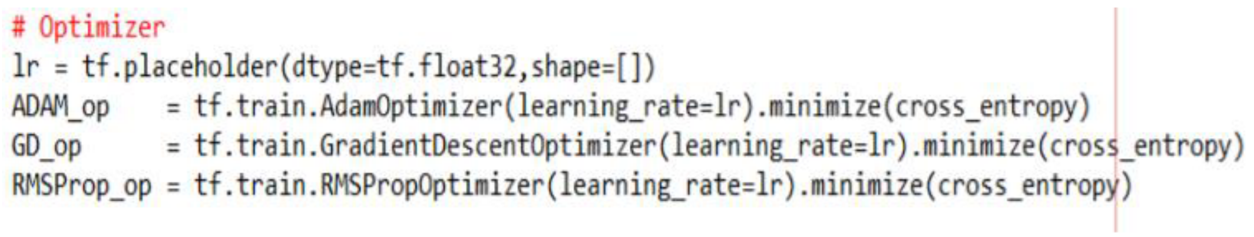
We also calculated the average message error rate of the mini-batch by comparing the receiver’s output on the element with the highest probability in the far end of the autoencoder defined by argmax.

```
# Loss function
cross_entropy = tf.losses.sparse_softmax_cross_entropy(labels=s,logits=s_hat)
```

```
# Metrics
correct_predictions = tf.equal(tf.argmax(tf.nn.softmax(s_hat),axis=1,output_type=tf.int32),s)
accuracy = tf.reduce_mean(tf.cast(correct_predictions,dtype=tf.float32))
bler = 1.0 - accuracy
```

Optimizing the 'Autoencoding' Algorithm:

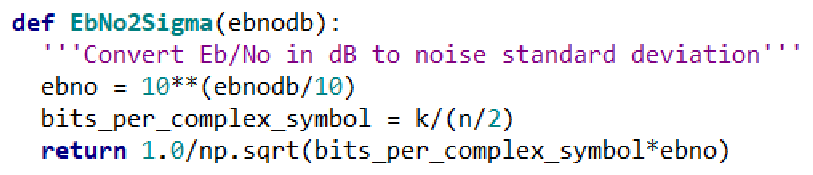
Here we define an optimizer algorithm that updates the weights of the autoencoder according to the current loss and the gradient of the batch. In order to bring out the best performance from our Autoencoder we need to test the results against different optimization settings. For this project we have chosen the Adam optimizer, Gradient Descent Optimizer and the RMSProp Optimizer to minimize our loss function and use a placeholder as learning rate to adjust it during training. In the end we will compare the Model's performance against different optimizer settings.



```
# Optimizer
lr = tf.placeholder(dtype=tf.float32,shape=[])
ADAM_op    = tf.train.AdamOptimizer(learning_rate=lr).minimize(cross_entropy)
GD_op      = tf.train.GradientDescentOptimizer(learning_rate=lr).minimize(cross_entropy)
RMSProp_op = tf.train.RMSPropOptimizer(learning_rate=lr).minimize(cross_entropy)
```

Defining a SNR Function:

We need this function to train our autoencoder at a desired SNR point. It simply calculates the noise standard deviation for a given SNR.



```
def EbNo2Sigma(ebnodb):
    '''Convert Eb/No in dB to noise standard deviation'''
    ebno = 10**(ebnodb/10)
    bits_per_complex_symbol = k/(n/2)
    return 1.0/np.sqrt(bits_per_complex_symbol*ebno)
```

Training the Autoencoder:

During all training epochs we set the SNR to 7.0 dB as training the autoencoder at a block-error-rate (BLER) of around 0.01 leads to a fast generalization. It's better to use a GPU here for faster training, one can opt the GPU mode in Google Colab or can just take aid from the CUDA.

First all the trainable variables need to be initialized. For this Tensorflow has already got functions where it uses the global variables initializer keeping all the weights ready. We need to make sure that our model doesn't learn the noise statistics. This adds up to the Neural Network's degraded generalization performance (A Neural Network does extremely well in capturing the underlying statistics of the training data).

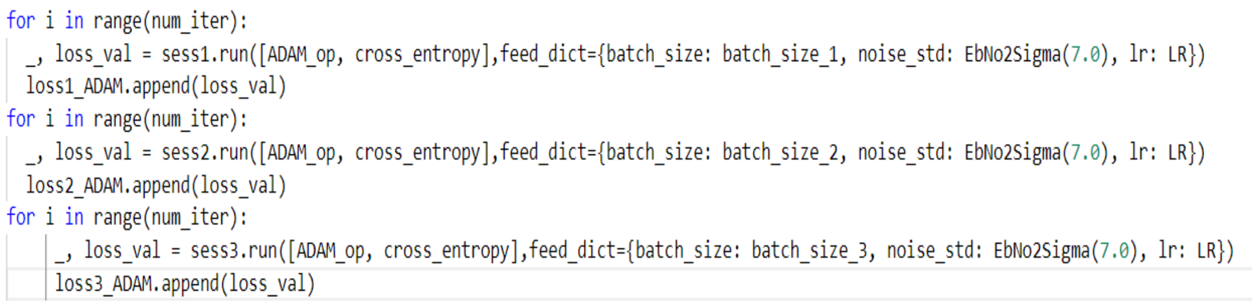
Note: We have defined nine different Tensorflow sessions to train our autoencoder with three different optimizers at three different tunings as described below :

1. 10,000 iterations of running the train_op function with a batch_size of 256 messages and a learning rate of 0.001.

2. 10,000 iterations of running the train_op function with a batch_size of 512 messages and a learning rate of 0.001.

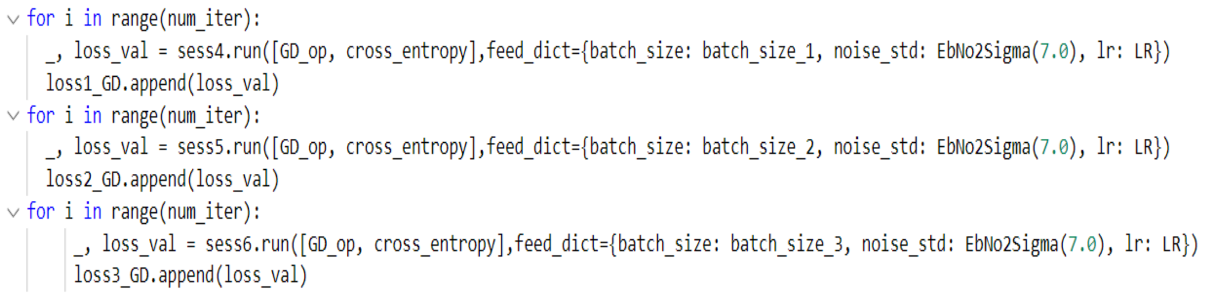
3. 10,000 iterations of running the train_op function with a batch_size of 1024 messages and a learning rate of 0.001.

<u>Training with Adam Optimizer:</u>



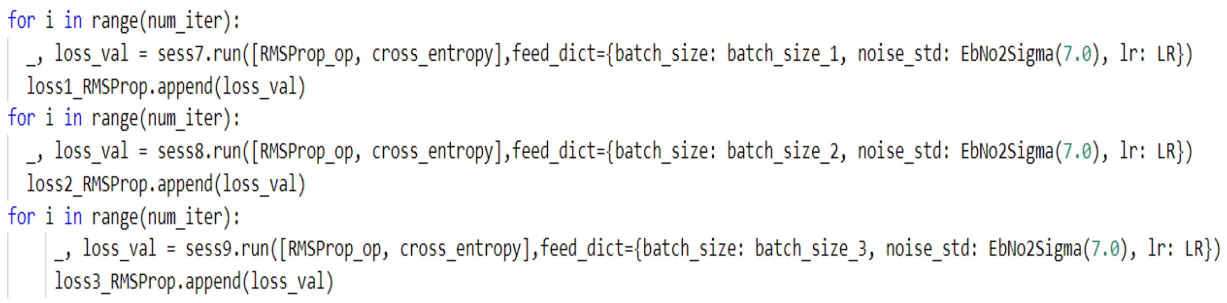
```
for i in range(num_iter):
    _, loss_val = sess1.run([ADAM_op, cross_entropy],feed_dict={batch_size: batch_size_1, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss1_ADAM.append(loss_val)
for i in range(num_iter):
    _, loss_val = sess2.run([ADAM_op, cross_entropy],feed_dict={batch_size: batch_size_2, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss2_ADAM.append(loss_val)
for i in range(num_iter):
    _, loss_val = sess3.run([ADAM_op, cross_entropy],feed_dict={batch_size: batch_size_3, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss3_ADAM.append(loss_val)
```

<u>Training with Gradient Descent Optimizer:</u>



```
for i in range(num_iter):
    _, loss_val = sess4.run([GD_op, cross_entropy],feed_dict={batch_size: batch_size_1, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss1_GD.append(loss_val)
for i in range(num_iter):
    _, loss_val = sess5.run([GD_op, cross_entropy],feed_dict={batch_size: batch_size_2, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss2_GD.append(loss_val)
for i in range(num_iter):
    _, loss_val = sess6.run([GD_op, cross_entropy],feed_dict={batch_size: batch_size_3, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss3_GD.append(loss_val)
```

<u>Training with RMSProp Optimizer:</u>

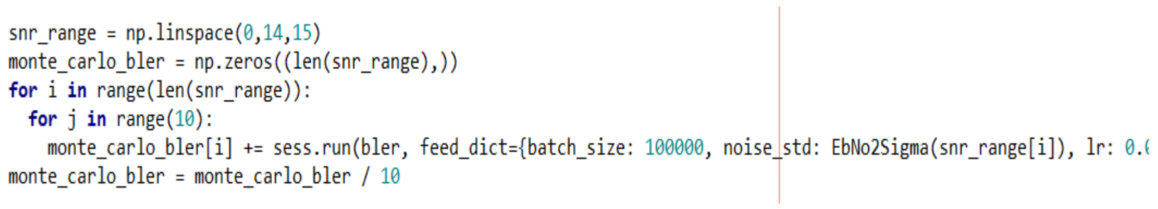


```
for i in range(num_iter):
    _, loss_val = sess7.run([RMSProp_op, cross_entropy],feed_dict={batch_size: batch_size_1, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss1_RMSProp.append(loss_val)
for i in range(num_iter):
    _, loss_val = sess8.run([RMSProp_op, cross_entropy],feed_dict={batch_size: batch_size_2, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss2_RMSProp.append(loss_val)
for i in range(num_iter):
    _, loss_val = sess9.run([RMSProp_op, cross_entropy],feed_dict={batch_size: batch_size_3, noise_std: EbNo2Sigma(7.0), lr: LR})
    loss3_RMSProp.append(loss_val)
```

<u>Evaluating the autoencoder :</u>

We need to run a Monte Carlo simulation to get an accurate BLER for each SNR point. In this example we simulate the BLER from 0 to 14dB by running 10 mini-batches of 10,000 messages for each SNR point.

Note: Monte Carlo Simulation, also known as the Monte Carlo Method or a multiple probability simulation, is a mathematical technique, which is used to estimate the possible outcomes of an uncertain event.


```
snr_range = np.linspace(0,14,15)
monte_carlo_bler = np.zeros((len(snr_range),))
for i in range(len(snr_range)):
  for j in range(10):
    monte_carlo_bler[i] += sess.run(bler, feed_dict={batch_size: 100000, noise_std: EbNo2Sigma(snr_range[i]), lr: 0.
monte_carlo_bler = monte_carlo_bler / 10
```

Performance of the autoencoder is measured by plotting its BLER vs SNR over a range of SNR. We plot the BLER vs SNR using matplotlib.

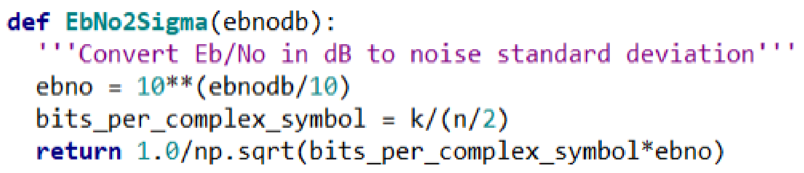
Channel capacity

Let's now try to compute the channel capacity for our Autoencoder communication system. This denotes the maximum information rate that is being achieved in our channel from source to destination. Before calculating that we need to go through some terminologies like:

Channel capacity theorem**: -** It is a theorem which relates channel capacity with Bandwidth and Signal to Noise Ratio. The Shannon capacity theorem defines the maximum amount of information, or data capacity, which can be sent over any channel or medium. It can be represented mathematically as: SNR can be calculated in Linear or dB format.

$$C = B\log_2\left(1+\frac{\mathrm{S}}{\mathrm{N}}\right)\mathrm{bits}$$

Decibels are logarithmic, not linear. Here we use the linear format. For this we create a function that gives the linear format for a given SNR in dB.

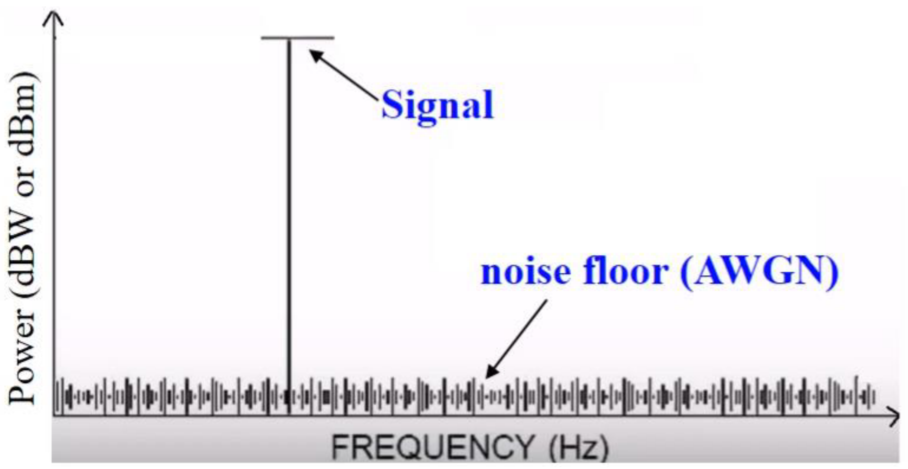

```
def EbNo2Sigma(ebnodb):
  '''Convert Eb/No in dB to noise standard deviation'''
  ebno = 10**(ebnodb/10)
  bits_per_complex_symbol = k/(n/2)
  return 1.0/np.sqrt(bits_per_complex_symbol*ebno)
```

The SNR for our project will be set to 7.0 dB as training the autoencoder at a block-error-rate (BLER) of around 0.01 leads to a fast generalization.

Bandwidth(B):- It is the difference between the lowest and highest frequencies that can pass through the channel. Analog signal bandwidth is measured in terms of its frequency (Hz) but digital signal bandwidth is measured in terms of bit rate (bits per second, bps). Note that greater the bandwidth of a channel, higher the data rate. Ideally channel should provide more bandwidth and signal must occupy less bandwidth.

Since we are working with digital data, we will consider the bandwidth to be 3100Hz.

Signal to Noise Ratio(S/N) - Compares the level of a desired signal to the level of background noise. It is the ratio of signal power to the noise power, expressed in decibels. Considering our Additive white gaussian noise channel, the noise in frequency domain can be represented as :



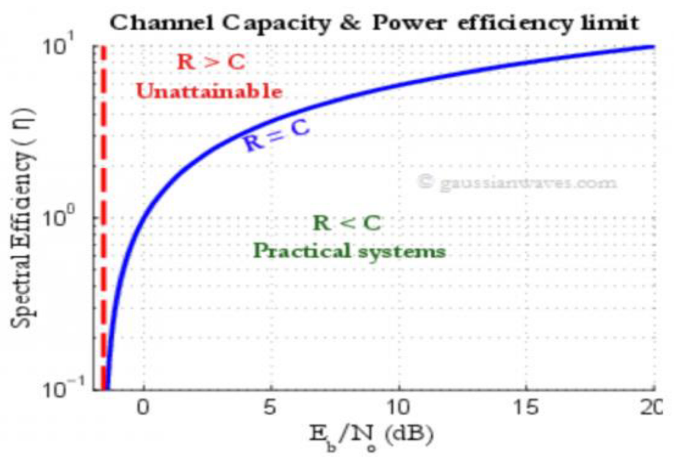
There are Two possible scenarios can occur for our case:

1. **Information rate < Channel capacity :**

In this case our message will be passed successfully through the channel even in the presence of noise.

2. **Information rate > Channel capacity :**

Here error-free transmission is not possible. Noise will be inevitably superimposed over our message. Signal will be lost in noise.



Now that we know these basic terminologies let's begin with calculation of channel capacity for our communication system. Therefore let us first create a python function that will determine the channel capacity for our system taking bandwidth of the signal in KHz and linear format of SNR.

```
import math
def calc_channel_capacity(B,SNR):
    a = (math.log((1+SNR),2))
    CC = B*a
    return CC

# Calculating Channel Capacity for 3100 Hz BW and 7.0dB SNR
channel_capacity = calc_channel_capacity(3.1,EbNo2Sigma(7.0))
```

Therefore for our Autoencoder system the Channel Capacity turns out to be 1.227 bits/second. This implies that our Information transfer rate must remain below 1.227 bits/second for error free transmission of messages.

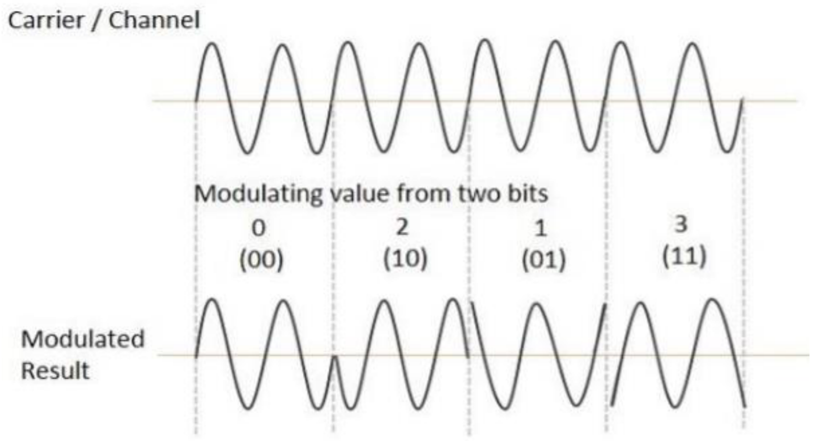
```
calc_channel_capacity(3.1,EbNo2Sigma(7.0))
1.2275948066851239
```

Modulation Schemes vs Autoencoder:

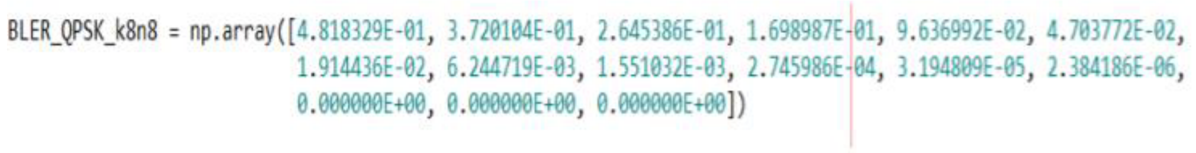
We need to understand the PSK scheme first, in Phase Shift keying modulation or PSK modulation phase of carrier is changed according to the digital data. It is digital modulation technique. It is used in broadcast video systems, aircraft and satellite systems. We use the following variations of the PSK scheme:

1. Quadrature Phase Shift Keying (QPSK):

It is a form of Phase Shift Keying in which two bits are modulated at once, selecting one of four possible carrier phase shifts (0, 90, 180, or 270 degrees). QPSK allows the signal to carry twice as much information as ordinary PSK using the same bandwidth



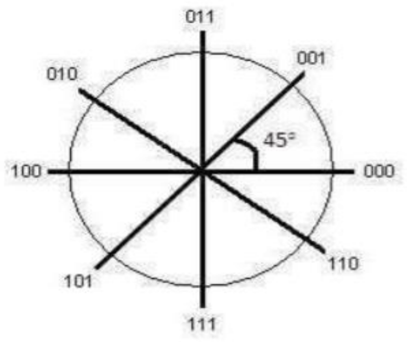
The uncoded BLER values of the quadrature phase shift keying (QPSK) modulation scheme that will be used to plot and compare it with the Autoencoder’s BLER.

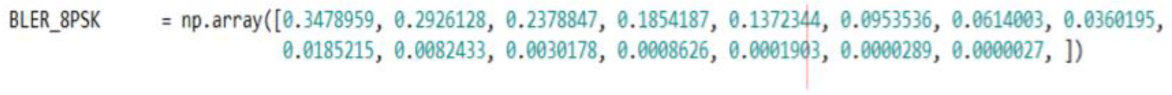


```
BLER_QPSK_k8n8 = np.array([4.818329E-01, 3.720104E-01, 2.645386E-01, 1.698987E-01, 9.636992E-02, 4.703772E-02,
                           1.914436E-02, 6.244719E-03, 1.551032E-03, 2.745986E-04, 3.194809E-05, 2.384186E-06,
                           0.000000E+00, 0.000000E+00, 0.000000E+00])
```

<u>2. 8-PSK:</u>

It is a multilevel PSK modulation which is a type of digital modulation based on carrier phase change. In 8-PSK eight different phase angles are used to represent bits. The constellation of 8PSK can be described as





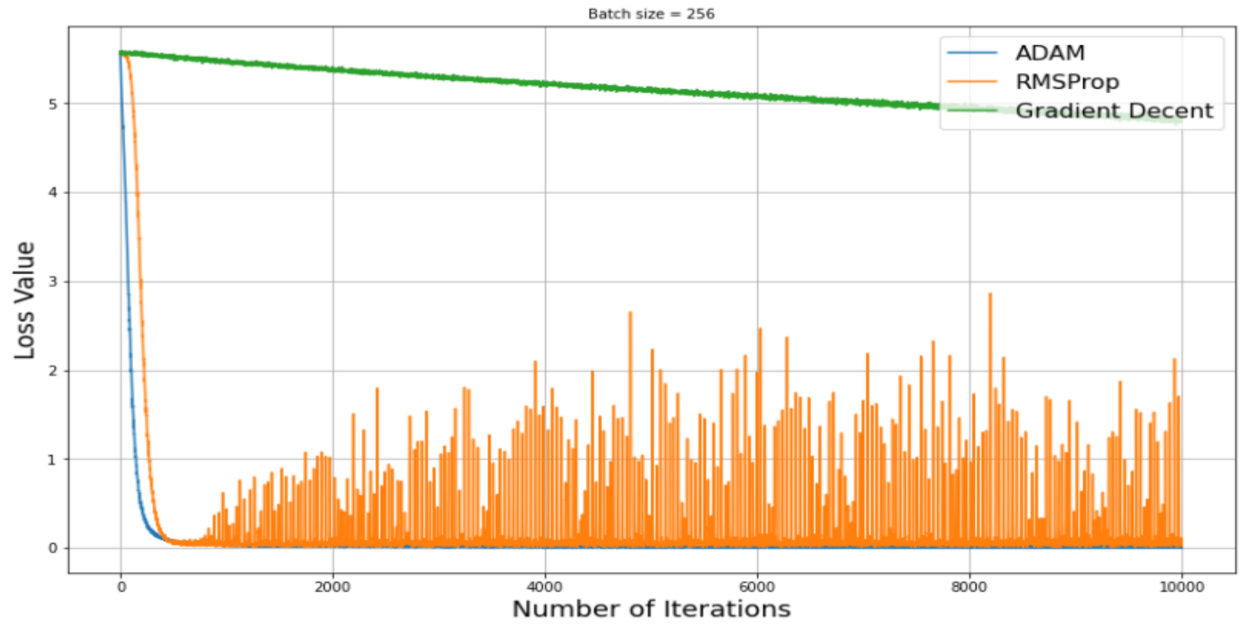
```
BLER_8PSK      = np.array([0.3478959, 0.2926128, 0.2378847, 0.1854187, 0.1372344, 0.0953536, 0.0614003, 0.0360195,
                           0.0185215, 0.0082433, 0.0030178, 0.0008626, 0.0001903, 0.0000289, 0.0000027, ])
```

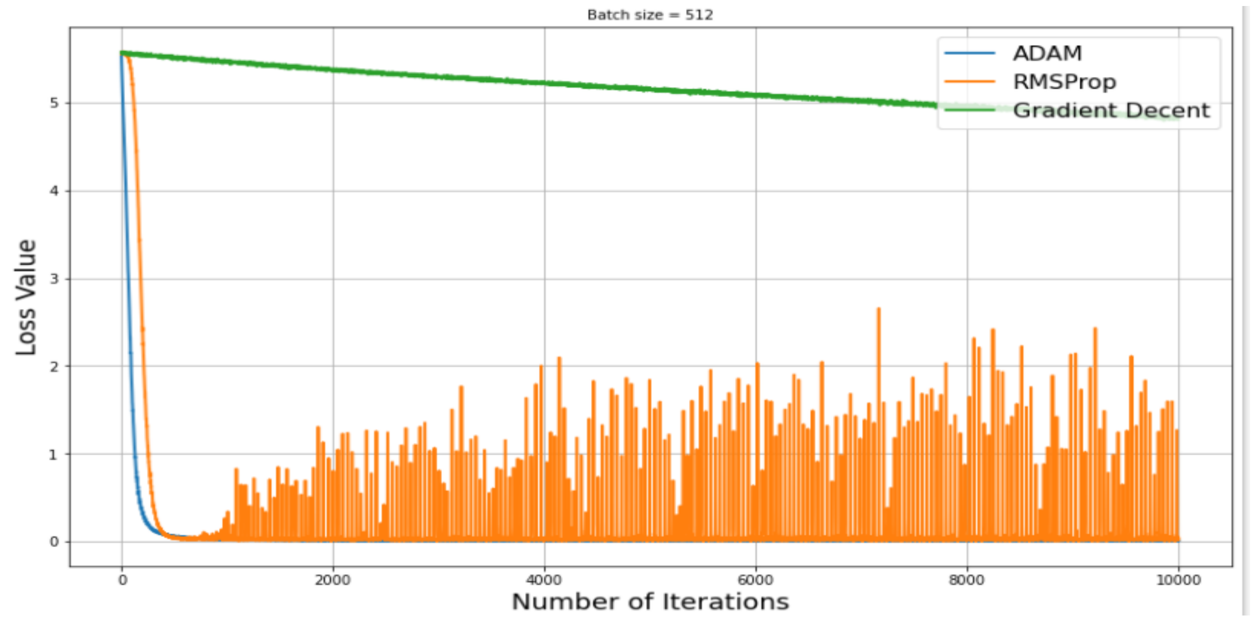
Performance of the Autoencoders is measured by plotting their BLER vs SNR over a range of SNR. We plot the BLER vs SNR using matplotlib.

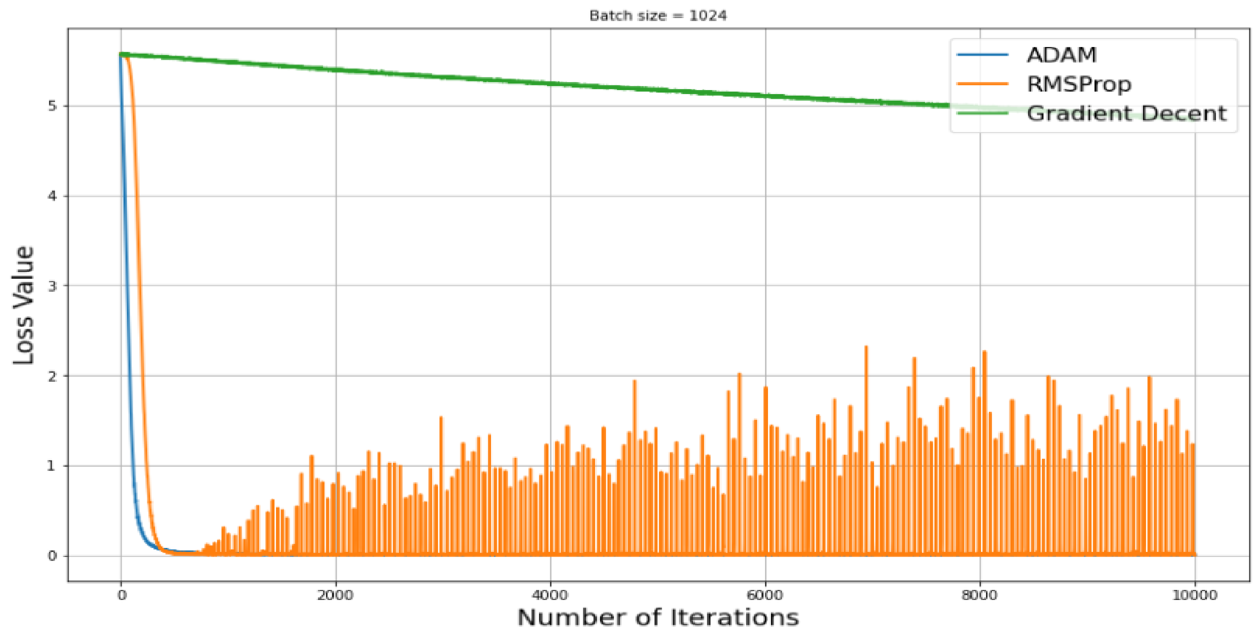
Results and Observations :

Comparing Losses among different Optimizer Settings:

We kept a track of the reduction in losses during the training of our model and below are their plot Comparisons for different Batch Sizes.





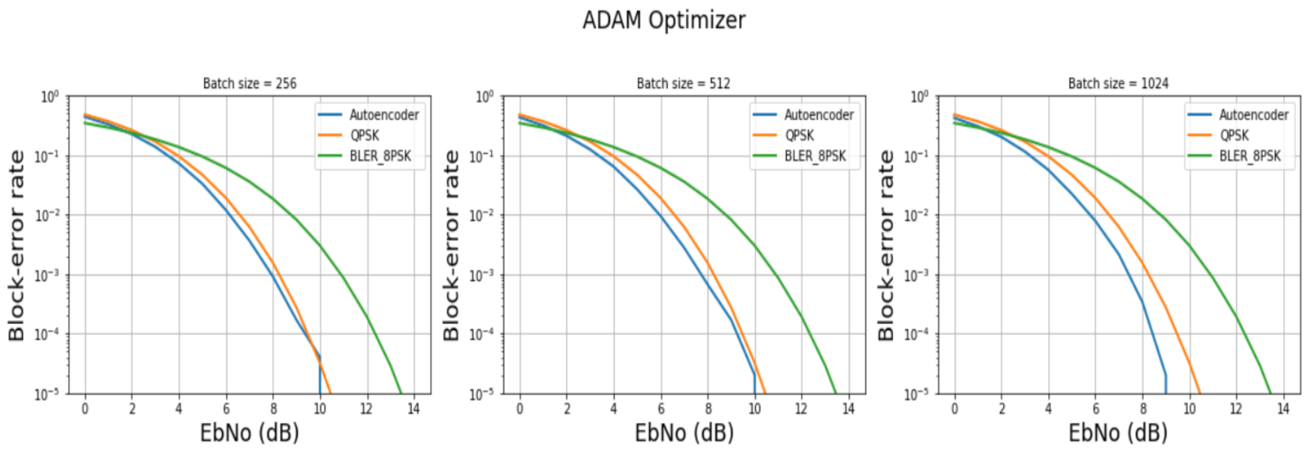


It is clearly evident that at for every batch size setting, Gradient Descent Optimizer shows a fairly slow learning rate whereas RMSProp learns good until a particular level where it starts heavily fluctuating, but ADAM Optimizer outperforms everyone.

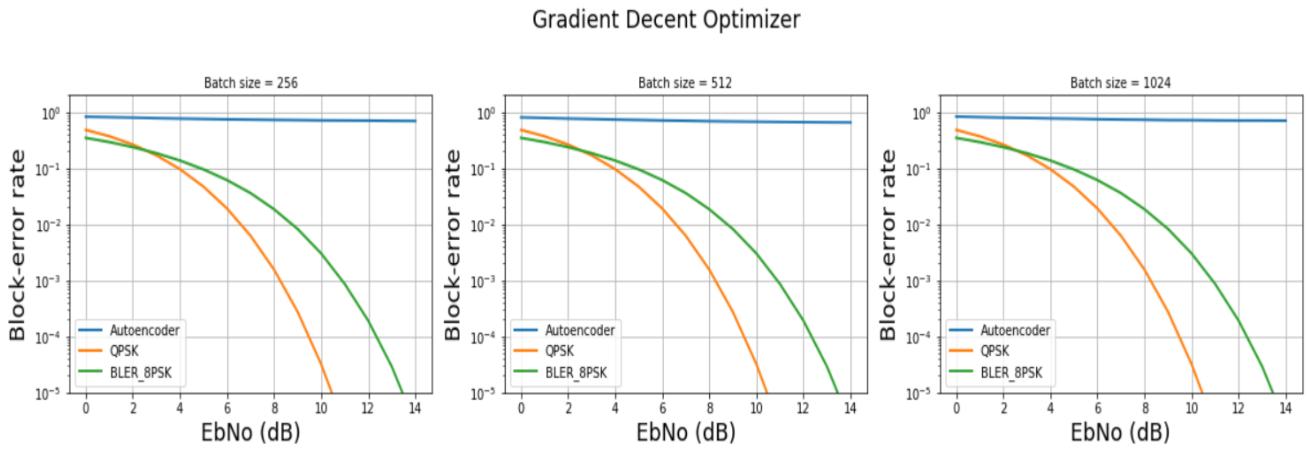
Comparing Autoencoder against Modulation Schemes:

Performance of the Autoencoders is measured by plotting their BLER vs SNR over a range of SNR. We plot the BLER vs SNR using matplotlib, Comparing it with other modulation schemes like Quadrature phase shift keying (QPSK) and 8PSK modulation:

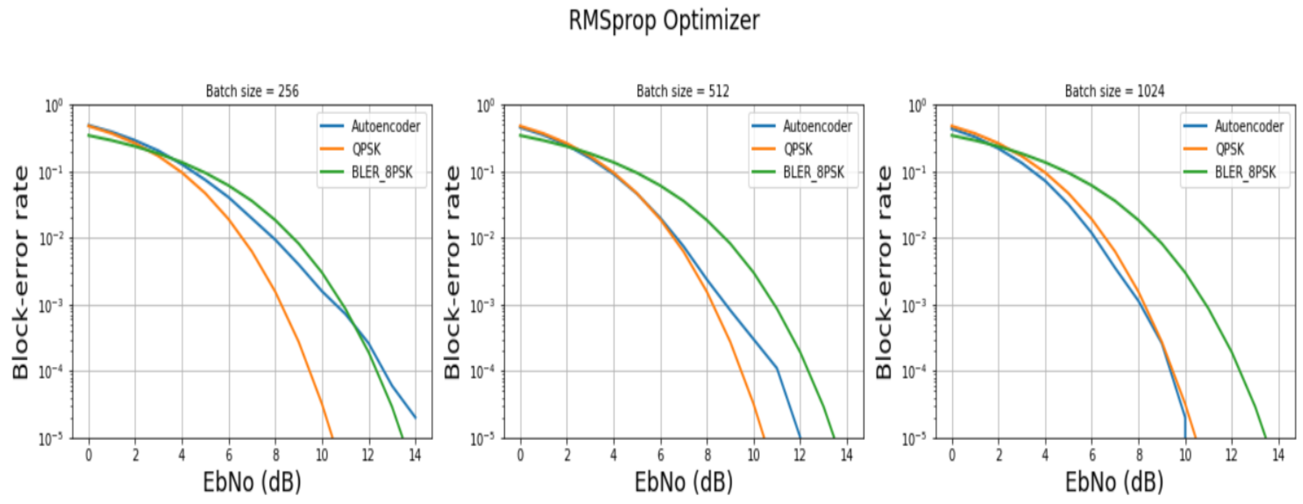
- **Autoencoder trained with Adam Optimizer:**



- **<u>Autoencoder trained with Gradient Descent Optimizer:</u>**



- **<u>Autoencoder trained with RMSProp Optimizer:</u>**



Observations:

- The autoencoder's BLER is lower than that of QPSK and 8PSK over the whole SNR range with ADAM optimizer, while RMSprop was lower only in higher batch sizes, and Gradient Decent was the worst in all cases.
- It can be seen that optimizing the encoder and decoder together is how we can force the autoencoder to extract only the features that are necessary and characterize the input data to store it in the bottleneck layer (i.e., where the smaller and dense representations are).

- We have observed that instead of optimizing the individual blocks of a conventional communication system (i.e., synchronization, symbol estimation, error correction, channel coding, modulation, etc.) we can use the autoencoder's optimization for the reconstruction loss.
- Future works in the field might include channel generalization by scaling from a simple AWGN model to more complex real-world channels.
- Regarding optimizers' performance, we noticed that:

Optimizer	**ADAM**	**Gradient Decent**	**RMSprop**
Learning	fast	slow	fast
Stability	stable	stable	fluctuating
Performance	good	Bad	good
Overall observation	best	bad	ok

Conclusion:

We have discussed the problem of unreliable communication and how solving it could be costly through constantly obtaining better equipment, and how new technologies like artificial intelligence can be a useful option which can be utilized effectively and save time and effort. Autoencoders is what has been tested in this project in different settings with three different optimizers. As unique problems need unique solutions, autoencoder has to be fine-tuned to suit a specific situation, so all of those hyperparameters set in this code are not fixed for every problem, some optimizers are good at certain situations and bad in others while. Other hyperparameters are good when increased but they could be computationally expensive. Eventually, there can be better tools than the one we used, but the question is how efficient is it? Because the ultimate goal is not merely to get good outcome, but to save time and effort at the same time. Thus, autoencoders remains – until the moment to say the least – one of the best tools in communication information.

References:

1. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift: https://dl.acm.org/doi/10.5555/3045118.3045167

2. Building Autoencoders in Keras : https://blog.keras.io/building-autoencoders-in-keras.html

3. He, K., Zhang, X., Ren, S., &amp; Sun, J. (2016). Deep residual learning for image recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). https://doi.org/10.1109/cvpr.2016.90

4. Learning to Communicate with Autoencoders: Rethinking Wireless Systems with Deep Learning : https://www.semanticscholar.org/paper/Learning-to-Communicate-with-Autoencoders%3A-Wireless-Morocho-Cayamcela-Njoku/d60ca7b32a768abb394438e9c7c9c29e146023b6

5. Learning to Communicate: Channel Auto-encoders, Domain Specific Regularizers, and Attention : https://arxiv.org/abs/1608.06409

6. Turbo Autoencoder: Deep learning based channel codes for point-to-point communication channels : https://arxiv.org/abs/1911.03038

7. Xu, J., Chen, W., Ai, B., He, R., Li, Y., Wang, J., Juhana, T., &amp; Kurniawan, A. (2019). Performance evaluation of autoencoder for coding and modulation in wireless communications. 2019 11th International Conference on Wireless Communications and Signal Processing (WCSP). https://doi.org/10.1109/wcsp.2019.8928071