Auto-encoder

Markus Søvik Gunnarsson

A report submitted for the course
MalCom - Machine learning for communications
Eurecom

April 2021

Abstract

This lab introduced hands-on experience about auto-encoders. The lab explored a lot of features regarding the auto-encoder; different constellation, activation functions, depth of mode and optimization. Practical experience in general machine learning, Python and Tensorflow/Keras have also been acquired. By following the assignment, there was a natural progression towards the obtained results and the discussion that followed.

Contents

Chapter 1

Introduction

1.1 Project Scope

This lab is an exprimental/practical exercise set given to students of the Malcom course at Eurecom to get more familiar with the auto-encoders, especially from a practical point of view. The content of the report is divided into two parts.

The first portion of the lab focuses on the understanding of the MQTT protocol, while the second portion focuses on implementing different features to the publisher and the subscriber.

1.2 Report Outline

The report consist of 2 main chapters.

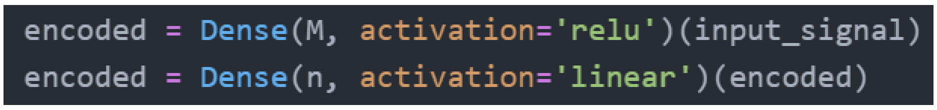
The report consist of an introduction (1) and a chapter for the assignment part of the lab (2). The last chapter includes everything from implementation, results and discussion.

Chapter 2

Autoencoders for End-to-End Communication Systems

2.1 a) Implementation of encoder and decoder.

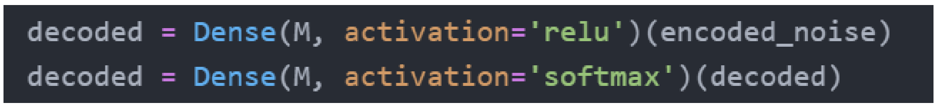
The autoencoder implemented by modifying the provided python code Lab1AutoencoderTODO.py.
The transmitter (encoder) is implemented as shown in figure (2.1)



```
encoded = Dense(M, activation='relu')(input_signal)
encoded = Dense(n, activation='linear')(encoded)
```

Figure 2.1: A snippet of a python code implementing a encoder.

From figure2.1 it is possible to see that the encoder consist of a input layer (dimension M), a dense layer with activation function "relu" (dimension M) and a dense/outputlayer with a linear activation function (dimension n). Where M is the number of messages to encode and n is the real-value signal.
The receiver (decoder) is implemented pretty similar. The screenshot is shown in figure (2.2)



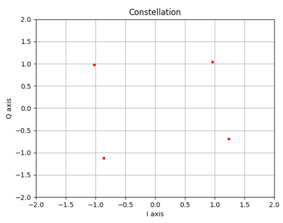
```
decoded = Dense(M, activation='relu')(encoded_noise)
decoded = Dense(M, activation='softmax')(decoded)
```

Figure 2.2: A snippet of a python code implementing a encoder.

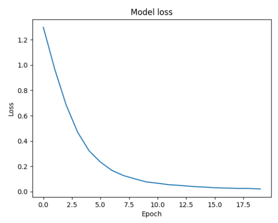
In figure (2.2) the decoder get a noisy input into a dense layer with an activation function "relu" (dimension M) and a dense/output layer with an activation function softmax (dimension M).

2.2 b) Verification and Constellation

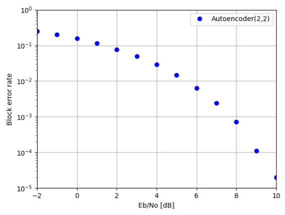
By running the newly implemented code for the autoencoder(2,2) for n=2 and k=2, where k is $log(m)^2$, the following outputs is generated (figure 2.3).



(a) Constellation diagram



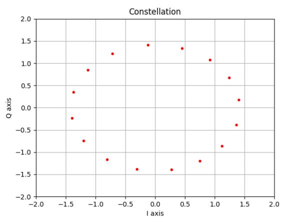
(b) model loss vs epoch



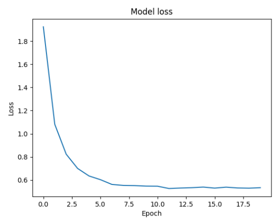
(c) BLER curve

Figure 2.3: Performance plots for autoencoder(2,2)

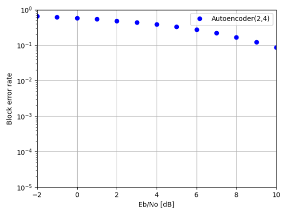
New performance plots, similar to figure (2.3), from an auto encoder(2,4) are shown in figure (2.4).



(a) Constellation diagram



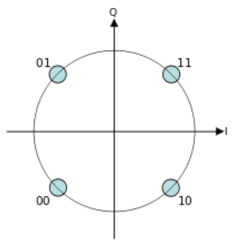
(b) model loss vs epoch



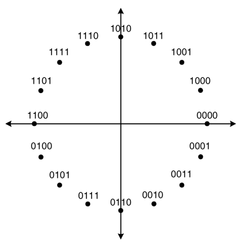
(c) BLER curve

Figure 2.4: Performance plots for autoencoder(2,4)

The learned constellations in figure (2.3.a) and (2.4.a) looks pretty similar to the known phase-shift-keyed modulated constellations of QPSK and 16-PSK. Pictures for reference are shown in figure (2.5)



(a) QPSK modulated constellation



(b) 16-PSK modulated constellation

Figure 2.5: Two different PSK-modulated constellation diagram

The learned constellation from the auto-encoders is all in all pretty similar to the reference constellations from figure 2.5. The slight difference in positioning is most likely due to introduced noise that induce some random and small phaseshift for each points in the constellation. This noise is gaussian distrubuted and introduced in the similated "real" channel.

2.3 c) Average power constraint

To introduce an Average power constraint (2.1)

$$E[|x_i|^2] \leq 1 \tag{2.1}$$

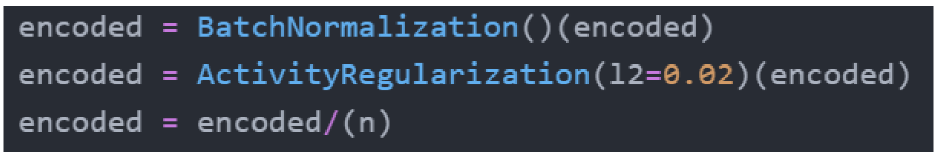
A python implementation for a finite case is needed (2.2)

$$\frac{1}{n}\sum_{k=1}^{n} |x_i|_k^2 \leq 1 \tag{2.2}$$

The squared sum can be rewritten into the L2 Norm.

$$\frac{1}{n}||X_i||_2 \leq 1 \tag{2.3}$$

The left side of the equation (2.3) can be implemented in the following way (figure 2.6).

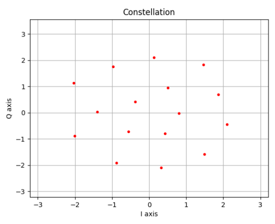


```
encoded = BatchNormalization()(encoded)
encoded = ActivityRegularization(l2=0.02)(encoded)
encoded = encoded/(n)
```

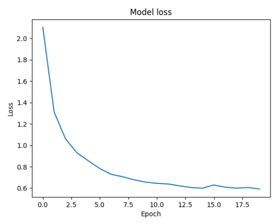
Figure 2.6: A snippet of a python code implementing an average power constraint.

Figure (2.6) shows that a batch normalization have been included to improve the results.

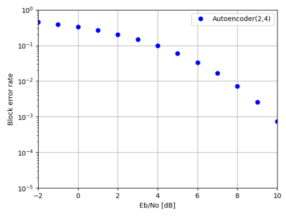
With this implemented the output of the new-improved auto-encoder(2,4) is displayed in figure (2.7)



(a) Constellation diagram



(b) model loss vs epoch



(c) BLER curve

Figure 2.7: Auto-encoder(2,4) with average power constraint.

In contrast to the old implementation, the new average power constrained gives a better Block error ratio (BLER). The model loss is the same as before. This is visible from the results in figure (2.7). A possible explanation for this is the increased length between constellation points in the new constellation diagram. A larger distance will make it easier to make a correct decision.

2.4 d) Activation function

By performing a batch normalization it seems like the linear activation function in the last layer in the encoder, can be swapped out with any of the following activation function without performance loss; tanh and sigmoid. Relu does not perform as good.

When swapping out the softmax activation function in the last layer of the decoder with another activation function, the only activation function that keeps the same performance is sigmoid.

By replacing the relu activation functions in the first layer in both the encoder and the decoder with the linear activation function, the performance will be kept somewhat similar. The model loss gets better in the linear activation case when the batch normalization is removed. A single layered auto-encoder with a linear activation function is very similar to PCA.

2.5 e) Depth of network

By removing one dense layer from both the encoder and decoder the BLER slightly increases for low SNR/bit/bit[-2 dB to 4dB]. This is opposite for higher values of

SNR/bit/bit [5dB to 10 dB]. The difference in BLER is all in all pretty low. The model loss is lower for the 2-layered network.

When increasing the depth of the network with 3 or 4 layers the performance goes down. A reason for this might be that the auto-encoder ends up being over-fitted since the dimension/amount of training data of the auto-encoder is rather low.

2.6 f) Performance of different AEs

The following list is a performance ranking in BLER of 3 different auto-encoders with 3 different SNR/bit/bit;

1. Auto-encoder(4,2) BLER = 0,24[-2dB] 0,13[0dB] 0,00000[10dB]
2. Auto-encoder(2,2) BLER = 0,25[-2dB] 0,15[0dB] 0,00005[10dB]
3. Auto-encoder(2,4) BLER = 0,66[-2dB] 0,58[0dB] 0,08000[10dB]

From the list it is possible to see that the worst performing auto-encoder is the (2,4). The other two is very similar to each other in terms of BLER(The (4,2) auto-encoder provides longer code-words than the (2,2) auto-encoder to little effect). If we look at the code rate R = $\frac{k}{n}$. We see that they rank after code rate.

1. Auto-encoder(4,2) R = $\frac{2}{4}$ = 0.5.
2. Auto-encoder(2,2) R = $\frac{2}{2}$ = 1.
3. Auto-encoder(2,4) R = $\frac{4}{2}$ = 2.

The code rate gives the proportion of the data that is useful. The auto-encoder is the (2,4) loses a lot of information as it transmits a total of 2 bits, while providing a total of 4 bits with important information.

2.7 g) Performance of different optimization techniques

The following list is a performance ranking of BLER with 4 different optimization techniques with 3 different SNR/bit/bit;

1. Adam BLER = 0,66[-2dB] 0,58[0dB] 0,088[10dB]
2. RMSProp BLER = 0,66[-2dB] 0,58[0dB] 0,094[10dB]
3. SGD BLER = 0,66[-2dB] 0,58[0dB] 0,099[10dB]
4. AdaDelta BLER = 0,88[-2dB] 0,87[0dB] 0,860[10dB]

Results from the list are pretty clear. Adam, RMSProp and SGD are pretty much equal in terms of performance. SGD is a little slower than the two other winners because it does not contain momentum calculation which increases the speed of optimizing. AdaDelta did not work at all. A reason for this can be that AdaDelta works based on adaptive learning rate per dimension, and there is not many dimensions to learn from in this particular case.