ENSEIRB-MATMECA
Telecommunications Semester 8

PR205 - Advanced Project

LOC
Learned and Optimized Communications

Students:

Salma Azzouzi, sazzouzi@enseirb-matmeca.fr
Yoan Echantillon,
yechantillon@enseirb-matmeca.fr
Drystan Groell, dgroell@enseirb-matmeca.fr
Chaimae Hamma, chamma@enseirb-matmeca.fr
Emma Huneidi, ehuneidi@enseirb-matmeca.fr
Mathis Le Gall, mlgall004@enseirb-matmeca.fr
Mateo Lopez, mlopez017@enseirb-matmeca.fr

Supervisor:

Romain TAJAN
romain.tajan@ims-bordeaux.fr

May 2022

Contents

Abstract

This paper presents the recent applications of deep learning for communication systems. It discusses the results of two chain models trained as autoencoders and optimized as an end-to-end reconstruction task. Additionally, a frame synchronization model based on Neural Network (NN) was added to the second chain to overcome the short length block used in continuous data transmission. In order to accelerate training and execution time, an interface between **Python** and **C++** was implemented.

1 Introduction

The main issue in communications is ensuring a reliable transfer of data. In order to do so, many channels and system models have been implemented with tractable mathematical models. In this regard, *AFF3CT* is a library that is used to implement the physical layer of most current and future standards (WiFi, 4G, 5G ...). Today's communication systems have been thoroughly optimized making them very efficient and stable. Introducing deep learning into such systems may not be as efficient in communications as it is in other fields such as speech recognition and computer vision. However, it is believed that it would have remarkable results in complex communications scenarios that are difficult to describe with mathematical modeling and analysis. Therefore, the aim of this project is to introduce deep learning into existing digital communication models using *PyTorch*, a machine learning framework. Simulation times in Python are slower than those in C++; to address this issue, an interface between Python and C++ was implemented. This paper first presents some deep learning basics, then discusses the results of two chain models trained as autoencoders (feedforward neural networks where the input is the same as the output) and presents the interface implemented between *Python* and *C++*. Finally, the last section is dedicated to the project management methodology.

2 Feedforward Neural Network

A feedforward Neural Network (NN), or multilayer perceptron (MLP), defines a mapping $y = f(x, \theta)$ and learns the parameters set θ. It is based on a number of L layers. Each layer carries out a specific mapping that depends on the output from the previous layer and on a set of parameters θ_l where $1 \leq l \leq L$. The main mapping used for layers in this paper is called *dense* or *fully connected*, which has the form

$$f(x, \theta_l) = \sigma(W_l x + b_l) \tag{1}$$

with $\theta_l = \{W_l, b_l\}$ where W_l is a matrix of weights and b_l is a bias vector. $\sigma(.)$ is an activation function that decides whether the neuron's input to the network is important or not in the process of prediction. The activation functions used in this paper are listed in Table 1.

Table 1: List of activation functions used

Name	$\sigma(x_i)$	Range
Linear	x$_i$	]-$\infty, +\infty$[
ReLU	max(0,x$_i$)	[0,+∞[
Softmax	$\frac{e^{x_i}}{\Sigma_j e^{x_j}}$	]0,1[

In general, NNs are trained using labeled data which contains a set of input-output vector pairs $(x_i, y_i^*), i = 1, ..., S$, where x is an input and y^* its desired output. The aim of the training is to minimize the loss function (equation 2)

$$L(\theta) = \frac{1}{S}\sum_{i=1}^{S} l(y_i, y_i^*) \tag{2}$$

considering the parameters in θ, where l is the loss function that compares the network's predicted output y_i with the corresponding label y_i^*. In this project, the cross entropy loss function was used (equation 3).

$$l(y_i, y_i^*) = -\Sigma_j y_{i_j} log(y_{i_j}^*) \tag{3}$$

After computing the loss function, the network can adjust the parameters in θ, using different algorithms, in order to produce accurate output values. The most known algorithm is stochastic gradient descent (SGD) which initializes the parameters θ randomly and then updates them iteratively as represented in equation 4.

$$\theta_{t+1} = \theta_t + \alpha \nabla \tilde{L}(\theta_t) \tag{4}$$

$\alpha > 0$ is the learning rate and $\tilde{L}(\theta_t)$ is an approximation of the loss function which is computed for a random *mini-batch*.

3 First Chain

Auto-encoders are the new generation of communication systems, they present the system as an end-to-end neural network (NN). An auto-encoder optimizes the transmitter and the receiver in one single process by learning the full implementation of the system. This first chosen communication system [1] is over an additive white Gaussian noise (AWGN) channel. Usually, the main goal of an auto-encoder is to transform the input to its low-dimensional alternative (representation) by eliminating the redundancy from the input. Contrary to this case, the "channel auto-encoder" seeks to recover the transmitted signal in the output with minimal error.

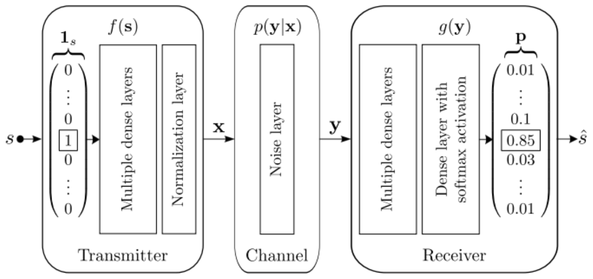


Figure 2: A transmitter, a receiver and an AWGN channel represented as a single auto-encoder

3.1 Network Model

A model of the auto-encoder is shown in Figure 2. The transmitter was considered as a feedforward NN with 2 dense layers and a normalization layer that ensured an amplitude constraint $|x_i| \leq 1 \forall i$ or an average power constraint $E(|x_i|^2) \leq 1$. The transmitter communicates one out of M possible messages where $M = 2^k$ (i.e k bits) through n discrete channel uses. Thus, the input s was coded as an M-dimensional one-hot vector 1_s where the sth element is equal to one and zero otherwise. The communication rate of this communications system is $R = k/n$ [bit/channel use].

As already mentioned, the channel was an AWGN with a fixed variance $\sigma = (2RE_b/N_0)^{-1}$, where E_b/N_0 denotes the energy per bit (E_b) to noise power spectral density (N_0) ratio. The receiver was also designed as a feedforward NN with 2 dense layers where the last one carried a *softmax* activation in order to generate a probability vector $p \in [0,1]^M$ over all messages as an output. Finally, the estimated message corresponds to the highest probability.

3.2 Training

In order to train the network model, labeled data was generated with a chosen *mini-batch* size. The training was carried out at a fixed value of $E_b/N_0 = 7dB$ over several *epochs*. In each one of them, the network's output of each *mini-batch* was computed then compared to the corresponding labels using the cross entropy loss function which is generally used in classification problems. Afterwards, the network's parameters were adjusted using *ADAM* optimizer with a learning rate of 0.001. The latter is an extension of the stochastic gradient descent (SGD) and is used to compute the adaptive learning rates for each parameter.

3.3 Results

3.3.1 Block Error Rate (BLER)

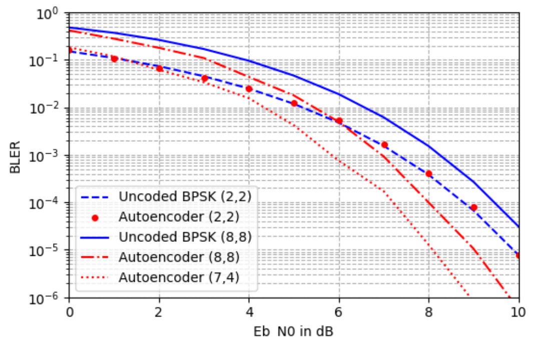
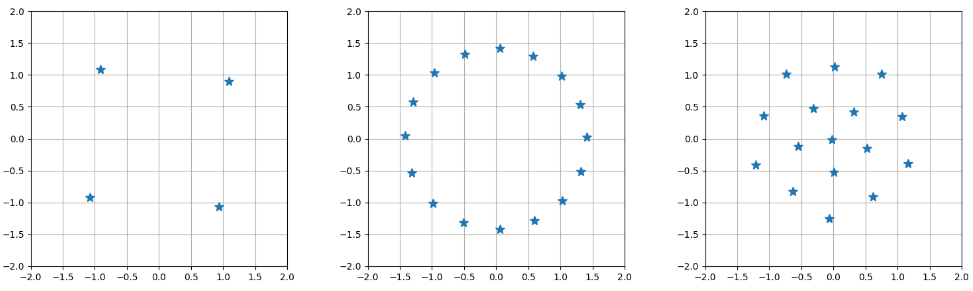


Figure 3: BLER versus Eb/N0 for the autoencoder and uncoded BPSK

Figure 3 compares the block error rate (BLER) of an uncoded communication system employing binary phase-shift keying (BPSK) modulation against the BLER achieved by the trained autoencoders (7,4), (2,2) and (8,8) with fixed energy constraint. The autoencoder achieves the same BLER as uncoded BPSK for (2,2), whereas it outperforms the latter for (8,8). Therefore, it attains a significant coding gain. Moreover, figure 3 shows that the autoencoder had the best performance for (7,4). However, it required more *epochs* and a greater *mini-batch* size during the training.

3.3.2 Constellations



(a) **(b)** **(c)**

Figure 4: Constellations for all messages produced by auto-encoders for different (n, k) : $(2, 2)$, $(4, 2)$ and $(4, 2)$ with an average power constraint

Figure 4 illustrates the representations of learned messages through different (n, k)-autoencoders. Figure 4a refers to (2,2) system which is equivalent to the existing quadrature phase shift keying (*QPSK)* with a random rotation. Figure 4b represents the $(4, 2)$ system which converges to a rotated *16-PSK*. The energy constraint forces the symbols to lie on the unit circle in order to maintain an equally spaced arrangement. Meanwhile, figure 4c reveals the previous system with a power average constraint. The latter displays the messages as equally spaced nearest neighbors within the unit circle.

4 Second Chain

In order to get closer to transmit and receive data in real-time, the second chain [2] includes synchronisations of time and frequency. As the first chain, it was developed as an autoencoder which can be divided into 3 parts: the channel which was implemented as a stochastic model and the transmitter and the receiver which were both neuronal networks.

The implementation of an autoencoder allows to make the training easier. In fact, the gradient of all layers can be calculated using the backpropagation algorithm. Nevertheless, it requires the knowledge of the channel mathematical model, which is why the two phase training strategy was used.

4.1 Two-Phase Training Strategy

The two-phase training strategy is based on transfer learning which consists in learning new tasks by using previously learned tasks. The two phases are :

- Phase 1: the training of the autoencoder with a stochastic channel model in order to approximate the channel behavior as near as possible. Unfortunately, it can create gaps with the real channel model, which is why a second phase is necessary to cope with this issue.

- Phase 2: the transmitter sends a certain number of messages through the real channel and the corresponding samples are linked to the receiver. These samples are, then, used as labeled data to oversee the receiver refining.

4.2 Chain Model

The transmitter was implemented as a neural network with linear and Rectified Linear Unit (ReLU) layers preceding a normalization layer.

As mentioned previously, a stochastic model was used to represent the channel. Figure 5 reveals that the signal undergoes diverse operations while it goes through the channel. Therefore, the signal was first upsampled by adding $\gamma - 1 = 3$ zeros after each symbol. Then, a root-raised cosine filter was used to filter and to create a random time offset $\tau_{off} \in [-T_s, T_s]$ as well. Next, while the signal was being modulated at a carrier frequency a random phase offset $\psi_{off} \in [0, 2\pi]$ was added. Finally, an AWGN block was used to represent the effects of the channel on the signal.

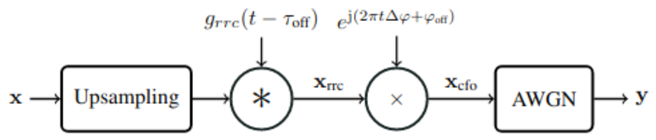

$g_{rrc}(t - \tau_{\text{off}})$
$e^{j(2\pi t\Delta\varphi + \varphi_{\text{off}})}$
$\mathbf{x}$
Upsampling
$*$
$\mathbf{x}_{\text{rrc}}$
$\times$
$\mathbf{x}_{\text{cfo}}$
AWGN
$\mathbf{y}$

Figure 5: Stochastic channel model

The receiver was also considered as a neural network with linear, ReLU and Softmax layers.

On top of that, 3 blocks were implemented to improve the chain performances: the Phase Estimator (PE), the Feature Extractor (FE) and the Offset Estimator (OE), which are all considered as dense NNs.

4.3 Training

The training of this chain was broken down into several steps. First of all, the entire chain was trained, according to figure 6.

This training process involved the creation of $2l + 1$ messages in parallel, the considered message being in the $(l + 1)^{th}$ position. All messages were then concatenated one after the other, to go through the channel. Subsequently, the slicing step recovered the considered message. This process finetuned the channel output. After a transition from complex to real, the feature extractor and the phase estimator steps were fulfilled. At the end, a new slicing was done. Finally, the outputs were concatenated to go into the decoder in order to get the estimated message. Following this first training, a new one was made while taking into account the offset estimator step, just after the passage through the channel so as to train the OE parameters.

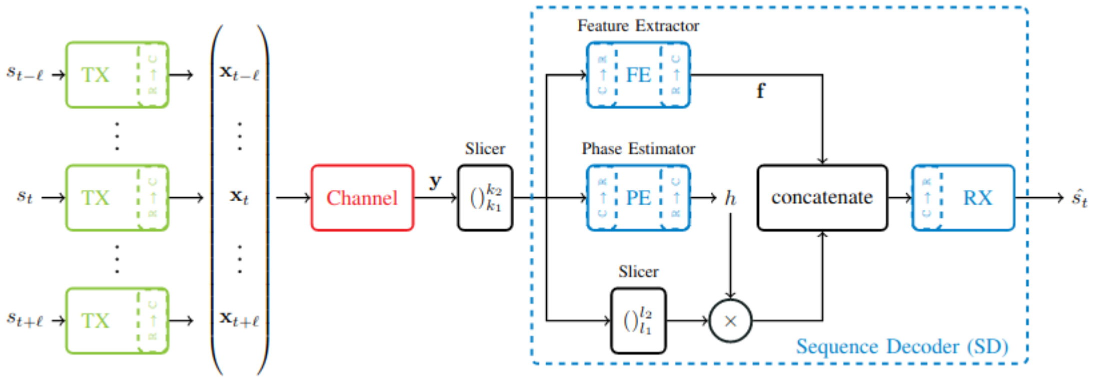


Figure 6: End to end training process

4.4 Results

The training with an ordinary laptop being very long (at least more than a week expected) there are currently no results produced such as BLER figures for example. The goal is to finish it soon and then to train the whole chain to produce some results and try to use the receiver part of the chain in a py_AFF3CT chain. That is why, some options to reduce training time are currently investigated. First, an idea could be to reduce the number of messages during the training but this strategy can lead to poor results. It could be also faster with the *C++* interface but this one is currently used on the first chain and not ready for the second one. This research will continue until the presentation day with the aim to get some results which can validate the efficiency of the chain model.

5 Python / C++ Interface

5.1 Basic Interface

The interface between *Python* and *C++* was implemented in two steps. The first was to use a *PyTorch* model to compile and execute a simple script, written in *Python*, in *C++*. To this end, the *Python* script was converted into a Torchscript, then loaded and executed in *C++*.
This method resulted in no gain of time, which was expected as the processing time of the conversion is too important before the time taken by the simple vector addition.

5.2 Advanced Interface

The second step was to adapt the previous method to the first chain. The entries also needed to be adapted so they could be easily adapted to the chain. A library *Loc* was developed to make and load torchscript models from *Python*.
This library was, then, used to load the different parts of the chain and their training without having to write anything else than *Python* code.
Thanks to this method, the execution was nearly twice as fast as normal *Python*, without using multi-threading (Table 2).

Table 2: Time of training and testing using *Python* or *C++* in seconds

Phase	Python	C++
Training (10 000 epoch)	128.2	73.1
Testing	–	–

However, due to the construction of the model, it did not allow the modification of variables after training. This means that the network needed to be trained again each time the model was tested with a different noise factor.

5.3 Fusion with py_AFF3CT

After being able to run our code in *C++*, despite writing it in *Python*, the main goal was to integrate our work in the py_AFF3CT library. This would allow the user to draw functions from aff3ct and only train the parts of the chain that interest them.

This was done by adapting an existing py_AFF3CT implementation, Pyaf, which uses functions from the library to generate, encode and modulate a signal, and afterwards process it with filters coded by the user.

Subsequently, this implementation was completed by adding a network based on the first chain, which would only train the receiver. It would simulate a receiver, receiving known data from an emitter and learning to decode it. In order to know if a such trained receiver can work in practice, further testing will be done. However, implementation of py_AFF3CT functions in the first chain is not finish yet because of the difficulties to understand pyaf and py_AFF3CT modules and also because there is no documentation available for these libraries, only examples of existing chain difficult to adapt to the project.

Another chain, this time with a trained emitter and a py_AFF3CT receiver, will be developed at the same time.
As those chains are not yet finished, the results are not yet available. When obtained, those results will be compared to both the first chain, and to a classical chain without machine learning.

5.4 Multi-threading

While running the code in *C++* already save time from *Python*, the most interesting time saving feature would be to multi-thread the training and testing of the network.

6 Project Management

6.1 Agile Methodology

Agile methodology was used during the project. Short-term goals were defined, and the final objectives had been adapted throughout the project, as it evolved. Besides, the project was broken down into several parts, in order to have a better vision and to facilitate problem solving. The planning of tasks and sprints for the coming weeks also helped to visualise the project's progress.

6.2 Roles and Responsibilities

A project manager was appointed at the beginning of the project. Subsequently, the project was divided into 3 sub-teams. First of all, one sub-team was responsible for the interface *Python*/*C++*. Meanwhile, 2 sub-teams were responsible for the implementation of communication chains.

6.3 Progress of Tasks and Milestones

The management tool *Trello* had been extremely useful in defining the tasks to be done, to be completed in the week, finished tasks, and also the *sprints*. At the beginning of each session, an update was made on what was done, on what should be done soon, and on the issues that hindered progress.

6.4 Conflict Management and Decision-Making Process within the Team

Communication within the team was good throughout the project. In fact, with the regular weekly meetings, there was real teamwork and the decisions were all taken together. When there were points of disagreement, discussions helped to find a solution. Therfore, there was no particular conflict.

6.5 Project Management Tools

Several tools were used to carry out the project. As mentioned before, *Trello* was a major tool for the management of the project, but other platforms were also useful. For example, to share code *GitHub* was used in order to make the work more efficient. In addition, to keep in touch within the team and with the supervising teacher and to plan meetings, *Discord* was used. Finally, *Excel* was used for planning tasks.

7 Conclusion

This project shed light on the use of deep learning in communication chains thanks to *PyTorch*. First of all, to get to grips with this machine learning framework, a first chain trained as an autoencoder had been implemented and the results were in line with what was needed. The interface *Python*/*C++* was implemented to compile and execute *python* scripts. Regarding the second chain, its implementation is almost complete. However, the training with an ordinary laptop being very long (at least more than a week expected) there are currently no results produced. This research will continue with the aim to get some results which can validate the efficiency of the chain model.

At the beginning of the project, the goal was to implement three chains, but time was missing and the first one required more time than expected, given that it was the first time that the group project was using *PyTorch*. This is why the initial objectives have been adapted in order to focus on the implementation of two chains, the interface and the use of py_AFF3CT.

References

[1] Tim O'Shea and Jakob Hoydis "An Introduction to Deep Learning for the Physical Layer" arXiv preprint arXiv:1702.00832v2, 2017.

[2] Sebastian Dorner, Sebastian Cammerer, Jakob Hoydis, and Stephan ten Brink "Deep Learning-Based Communication Over the Air" arXiv preprint arXiv:1707.03384v1, 2017.