NOISE CANCELLING USING DEEP LEARNING

Au513 - Prototypage rapide

BLIDI Nayel – DESCHAMPS Léo – MAZURIE Corentin - POISSONNET Clément – PRIEM Anthony

Table of Contents

I- Introduction

Telecoms are everywhere, and its wide use is one of its challenges. If all messages were sent on the same frequency, the large number of signals would make any transmission totally unreadable. To overcome this, telecoms broadcasts use encoding and frequency modulation of its signals.

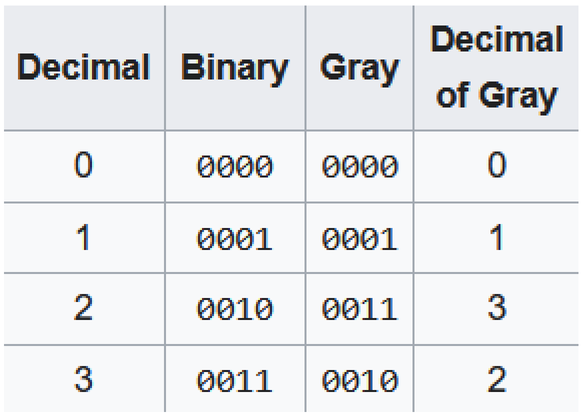
Because of the nature of emission and reception systems, these operations are computed using electronic components and logic of high complexity, that scale exponentially in size when handling more complex signals.

Hence the following project: by using deep learning for noise cancelling and messages decoding in embedded systems, this proof of concept aims at deciding whereas this solution may be relevant, as accurate and more efficient than the current methods.

II- State of the art

a) Encoding

The method of encoding we used for our simulation to compare with our own result is called Gray code. It is an ordering of the binary numbers such that two successive values differ in only one bit



Decimal	Binary	Gray	Decimal of Gray
0	0000	0000	0
1	0001	0001	1
2	0010	0011	3
3	0011	0010	2

We can see that the binary representations of 2 and 3 are inverted in the Gray representation, so 1 and 2 differ only by the second bit. As the transmissions are made by sending symbols (group of bits), it is useful to have this proximity between them. Indeed, the most used techniques of correction can only correct single-bits-errors.

b) Decoding

The most common technique to decode messages after their transmission is the MAP (Maximum A-Posteriori). This criterion is based on the search for an optimal threshold that minimizes the probability of error. It uses the a priori probability and the a posteriori probability of the symbol which is calculated relatively to the received value of the signal, by applying the Bayesian rule.

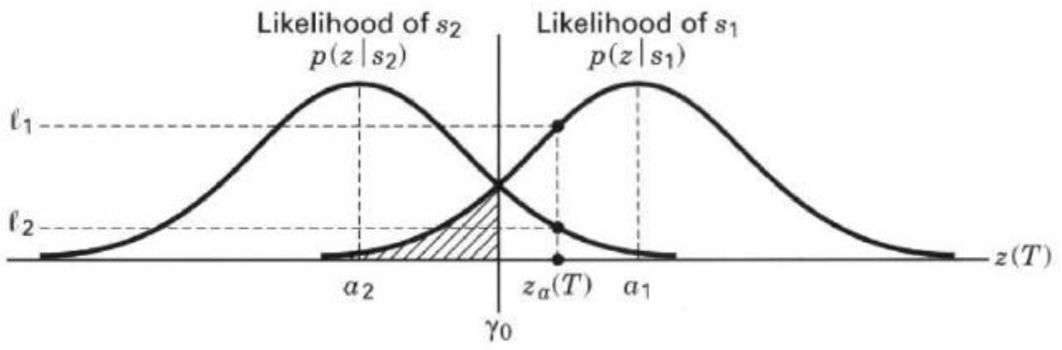


Figure – Conditional probability density functions : $p_z(z|s_1)$ and $p_z(z|s_2)$

c) Modulations

Binary phase shift keying (PSK): In this type of modulation the parameter of the carrier that varies according to the input information to the modulator is the phase. The pair of signals s1(t) and s2(t), used to represent the binary digits ”1” and ”0”, respectively, are defined as follows:

$s_1(t) = A\cos(2\pi f_c t), \ 0 \leq t \leq T$ if ”1”
$s_2(t) = A\cos(2\pi f_c t + \pi) = -A\cos(2\pi f_c t), \ 0 \leq t \leq T$ if ”0”

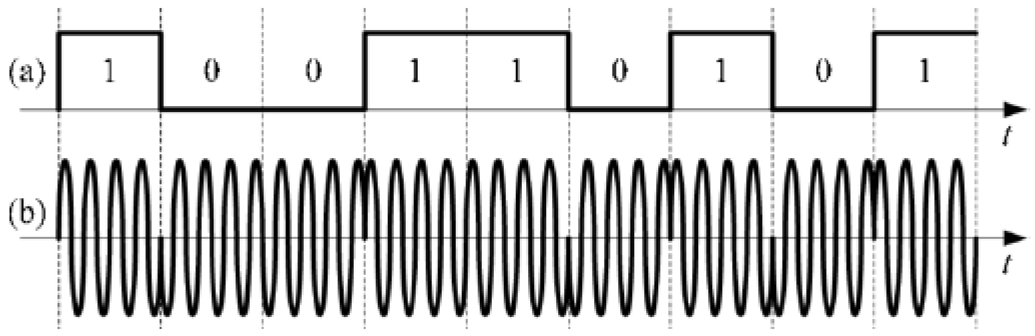
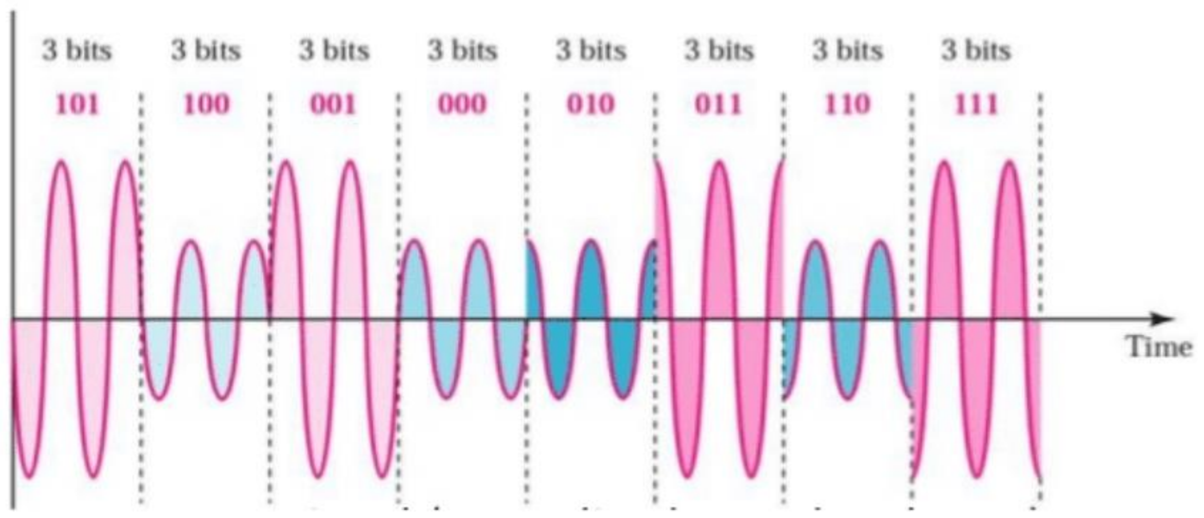


Figure – BPSK waveform.

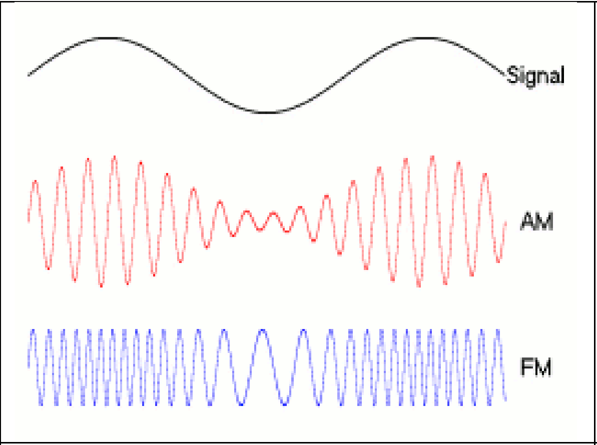
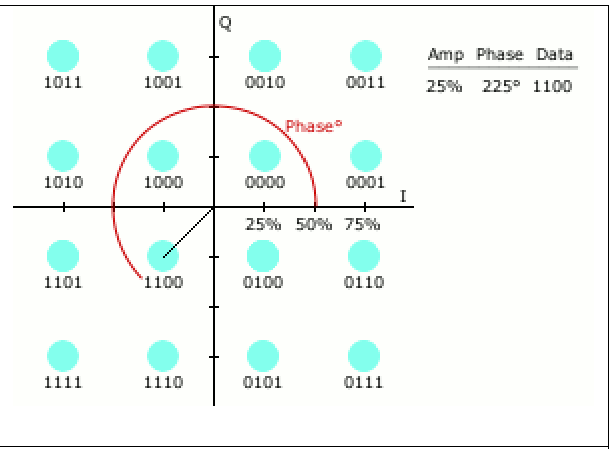
On this figure we can see the form of a cosine for each bit, with a phase shift of π for “0”. This modulation is very resistant to the noise because the bits are spread out in the complex plan, but very slow because we transmit bit one by one.

Quadrature Amplitude Modulation (QAM): In this modulation, the carrier varies according to both the amplitude and the phase.

I



In this example, we see a signal modulated with an 8-QAM. The bits are transmitted by symbols of 3 bits. There are 4 phases and 2 amplitudes, so 8 possibilities to represent every combination of 3-bits. This modulation can be very fast because we can increase the size of the symbols. For example, the 4096-QAM is used for television or 5G, and we transmit symbols of 64 bits.

	
AM and FM carrier frequency modulation	QAM symbols in the complex plan

III- Simulation

In order to fulfil the goals of the project, the previously presented steps had to be reproduced in a python environment to:

- Compare the results to the theoretical simulation
- Generate training and testing datasets to feed to the deep learning networks
- Process some of the data

Thus, we developed four classes, to tackle one encoding method each, and to make it scalable for further improvements.

a) PSK & BPSK-light

The PSK.py file handles the Phase Key Shifting encoding methods, as described above, by applying succeeding transformations to a binary message through a pipeline of methods. The class is initialized by a handful of parameters, such as the message length, carrier frequency, sampling frequency… but also extra parameters such as the time discrepancy used during the simulation.

Because the neural network is trained to recognise patterns rather than values, the datasets are generated using small values that improve the performances of the simulation, while keeping the proportionality of the constants.

The parameters used for all the following simulations, plots, datasets and models are defined as follows:

- Fc = 4 Hz (carrier frequency)
- Fs = 20 Hz (sampling frequency)
- Rb = 1 Hz (bit rate)

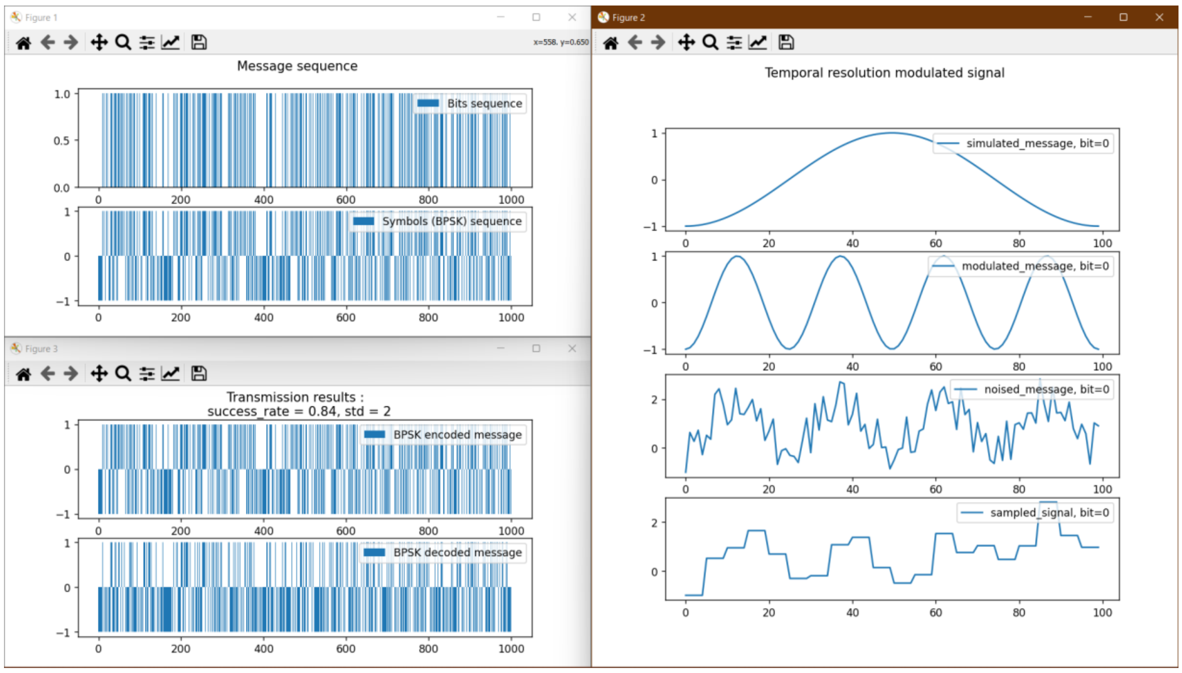
Although these values are unrealistic, they do not matter in the training of the neural network. As such, the following real values will be used latter on:

- Fc = 400Hz
- Fs = 8kHz (will be down sampled to 2kHz for compatibility with the neural network model)
- Rb =125Hz

The BPSK pipeline is represented as followed:

- BPSK.sequenceGenerator: generates a message as a sequence of bits (Figure 1, top subplot)
- BPSK.grayMapping: maps the message using Gray mapping (not applied)
- BPSK.BPSK: encodes the message as 1 and -1 symbols (Figure 1, bottom subplot)

- BPSK.signalModulation: represents the message as a cosine (Figure 2, first subplot) and modulates the carrier frequency accordingly (Figure 2, second subplot)
- BPSK.gaussianNoise: adds a random noise f standard deviation sigma (Figure 2, third subplot)
- BPSK.signalSampling: samples the noisy signal every 1/Fs seconds (Figure 2, fourth subplot)
- BPSK.sequenceDecider: simulates the map decoding operation by comparing it to a sin or cosine integral (Figure 3, top is real message, bottom is decoded message)
- BPSK.grayDemapping: maps the bits into its real binary representation (not applied)



BPSK Figures 1-2-3 and its subplots

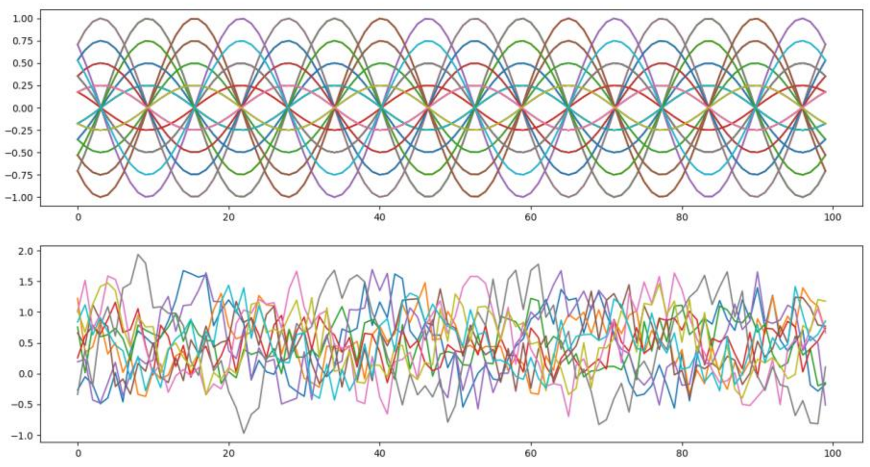
The BPSK-light follows the same step order, but instead of simulating a time signal, it operates all the transformation described above solely on the values that will be sampled in the end, without computing “useless” in-between values.
Because the main purpose of the BPSK-light is to create a training dataset, there is no need to simulate the decoding either, thus resulting in the following steps:

- BPSK_light.sequenceGenerator (directly generates the symbols instead of the bits message)
- BPSK_light.signalModulation
- BPSK_light.gaussianNoise

b) QAM

The QAM.py file has the same purpose as its PSK counterpart but applies a different encoding method that results in a higher overall bit rate speed, by transmitting more information every pattern.

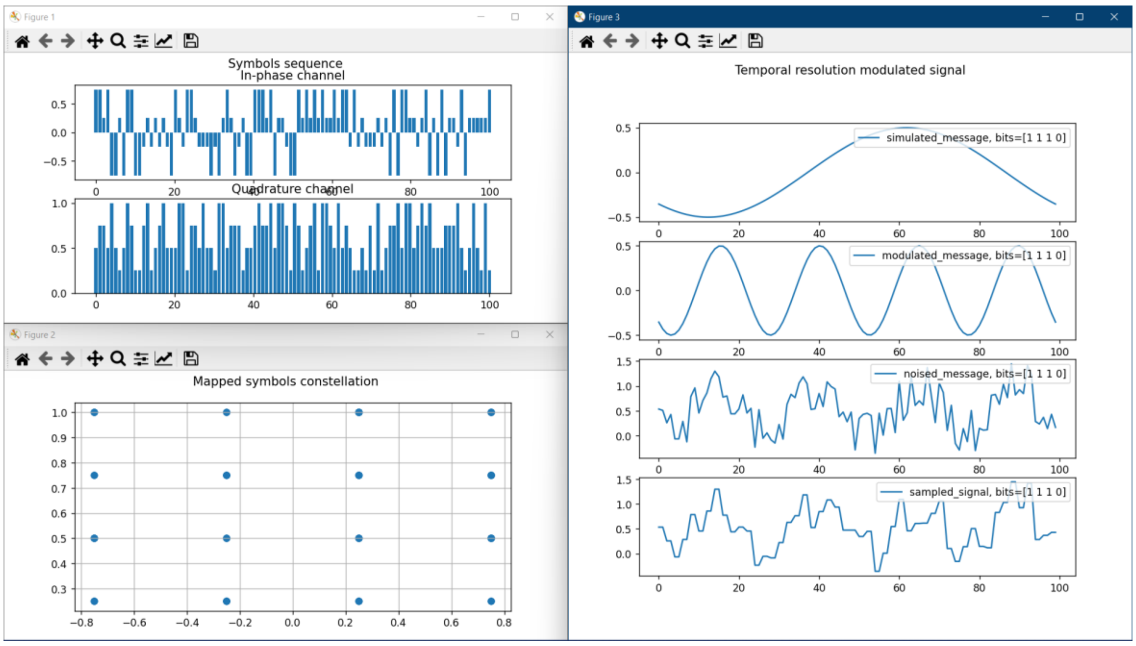
As shown below, the 16QAM method encodes 4 bits in a row (as explained in part II) by modulating both phase and amplitude of a signal.



Every 16QAM pattern, and its noisy real signal

The general structure is the same those in the PSK file, but the following points should be noted:

- The in-phase channel represents the symbol to code in the phase of the signal (Figure 1, top)
- The quadrature channel represents the symbol to code in the amplitude of the signal (Figure 1, bottom)
- These two channels can be spread out in a constellation plot (Figure 2)
- The simulation steps are the same as above (Figure 3)



16QAM Figures 1-2-3

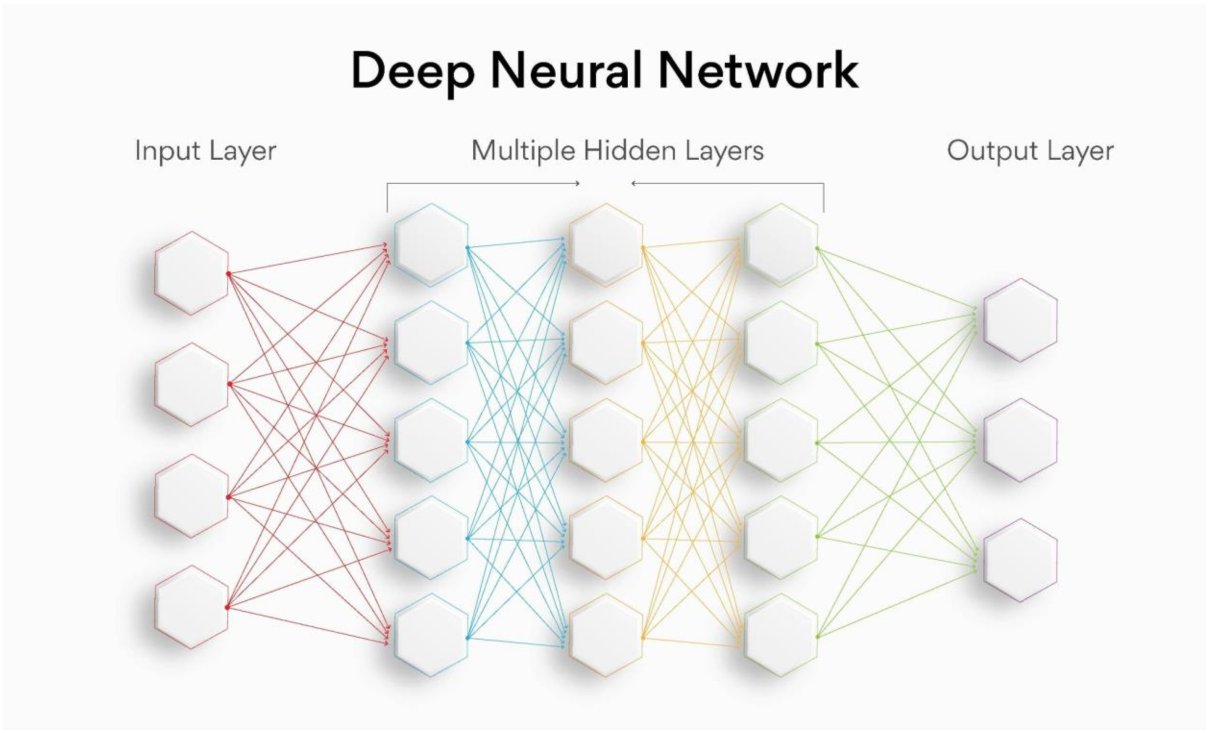
IV- Deep learning

a) Neural network models

In the continuation of the project, we developed neural network architectures using the Python programming language. Two specific models were created, namely "LayeredNN" and "SequentialLayeredNN."

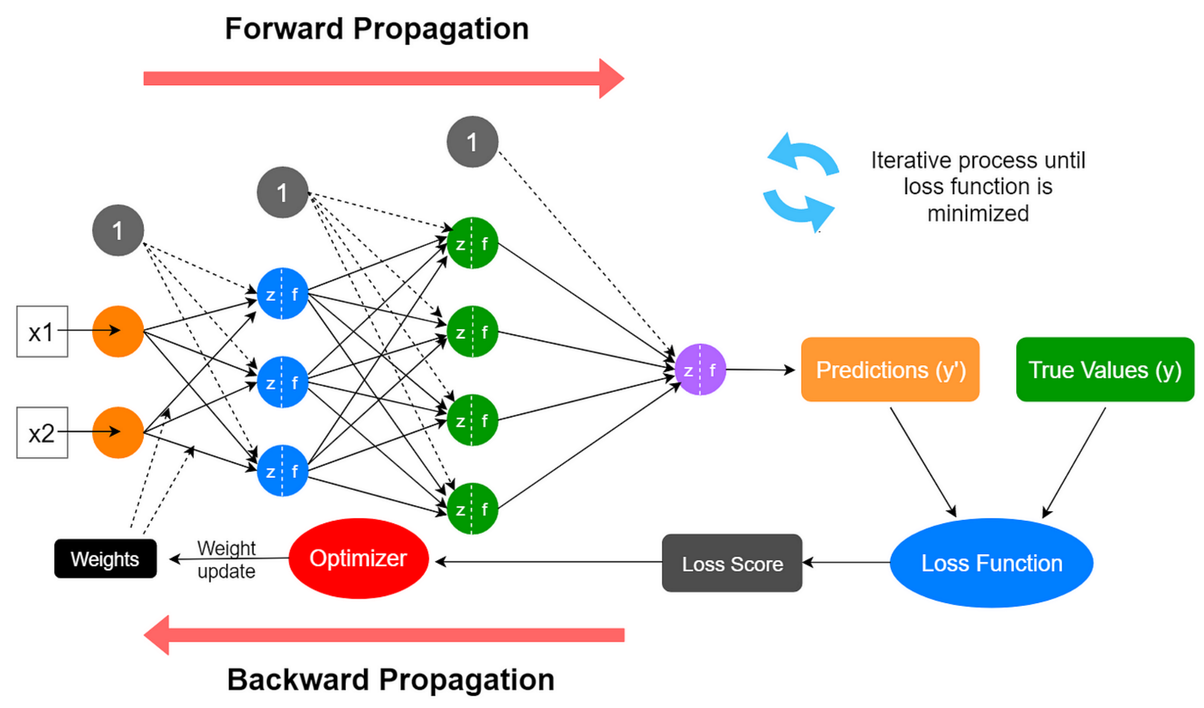
"LayeredNN" represents a particular type of neural network based on a layered architecture. It consists of four sub-layers, comprising 128, 64, 32, and 16 neurons, respectively. Each hidden layer progressively reduces dimensionality, thus favouring the extraction of complex features.

"SequentialLayeredNN" follows a similar approach but has a more condensed organization of layers. This model consists of three hidden layers with 128, 128, and 64 neurons. Additionally, it incorporates a random neuron deactivation function with a 1% probability in the second sub-layer, acting as regularization to enhance the model's robustness to new examples.



After defining the models, the training phase begins, allowing the user to choose hyperparameters such as the criterion, optimizer, and the number of epochs. These parameters influence the performance of the neural network.

Subsequently, the script generates training data using the functions mentioned in the previous section. The models are then trained on this data, and the results are presented in the form of graphs illustrating the evolution of training loss over epochs.

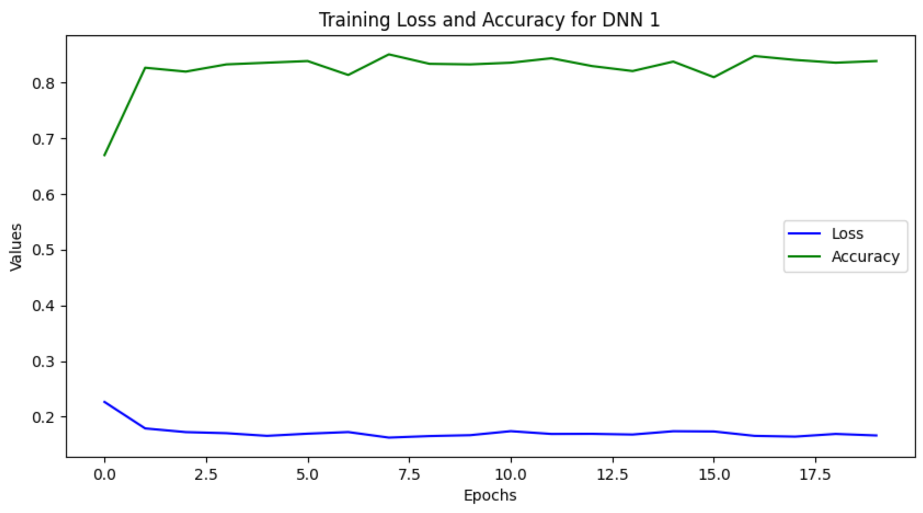


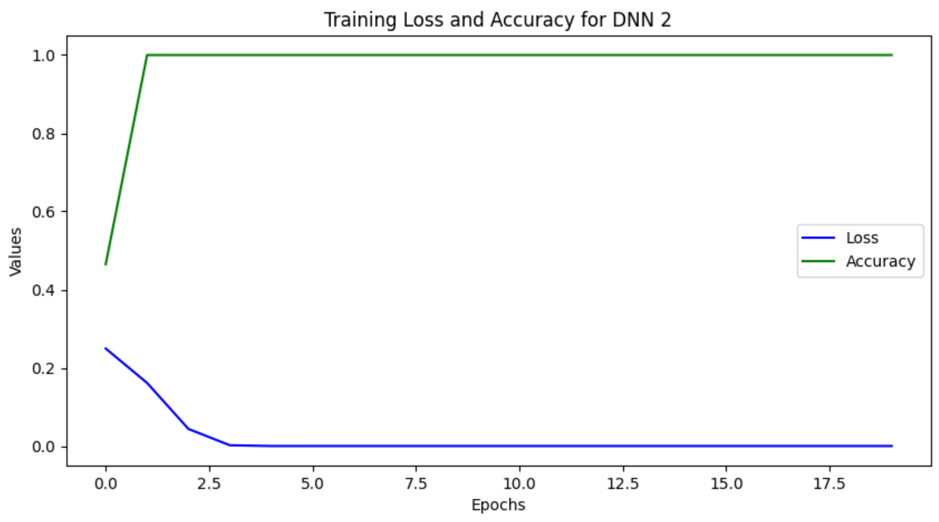
Finally, the models are evaluated on test data generated similarly to the training data. The outputs of the neural networks are compared to the expected data to assess the accuracy of the models.

b) BPSK results

	LayeredNN	SequentialLayeredNN
Correct message (on 10 000)	9620	9934
Percentage error	3.80%	0.66%
Percentage accuracy	96.2%	99.34%

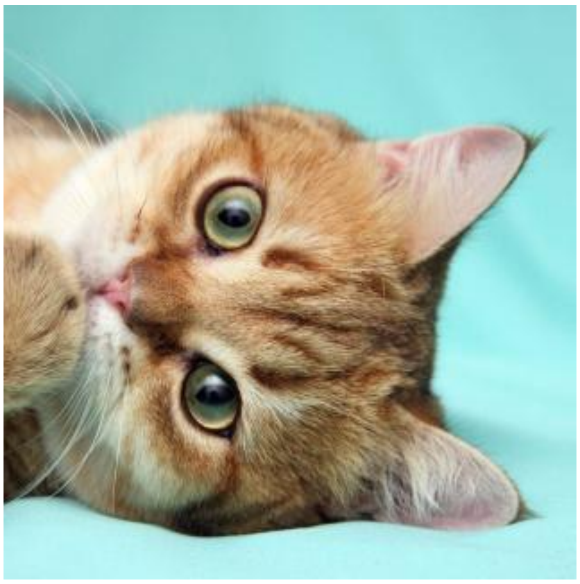
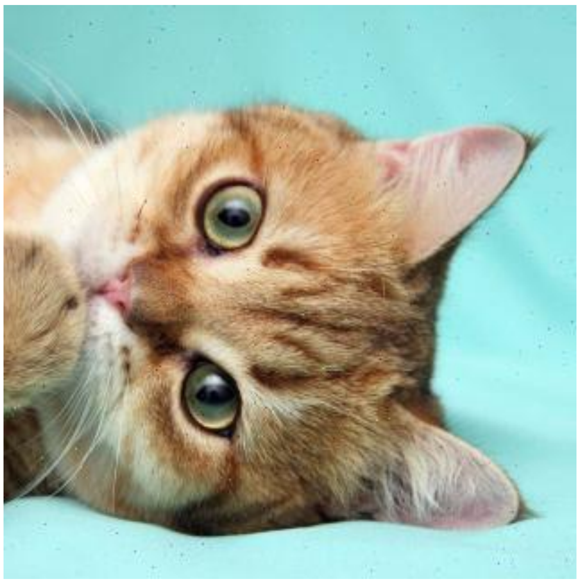
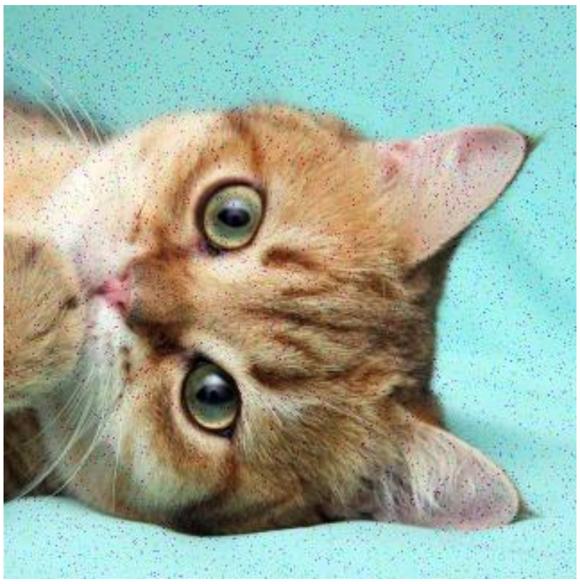
For this type of modulation, we have great results for the two models. These results suggest that both neural network architectures were successful in learning to decode messages efficiently, with the SequentialLayeredNN showing slightly superior performance. The high accuracy rates indicate the models' ability to generalize and perform well on test data, which is encouraging for their use in real-world communications applications.



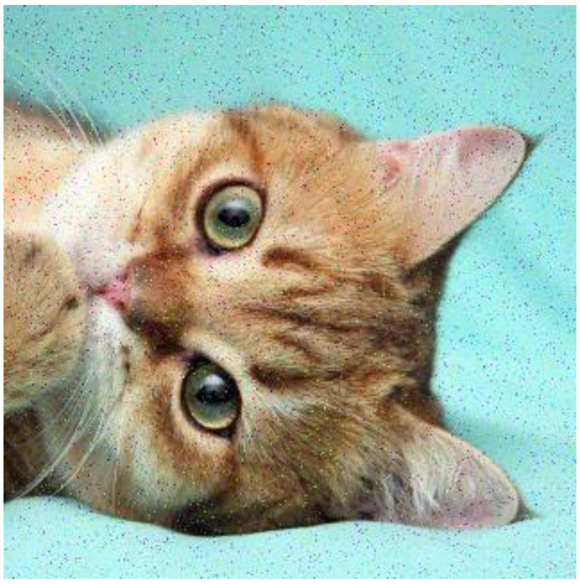
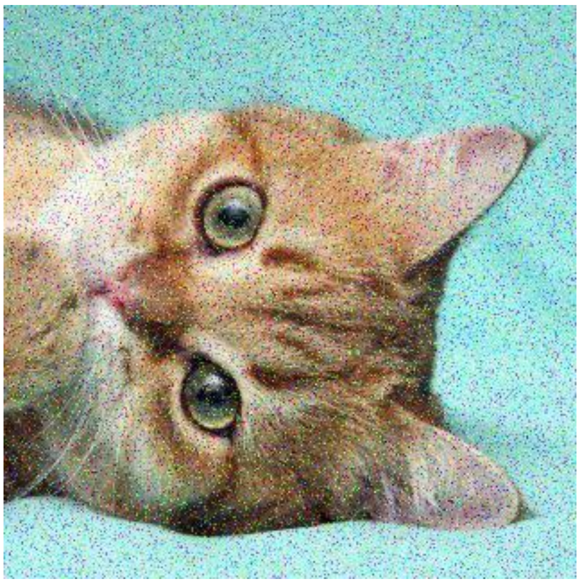
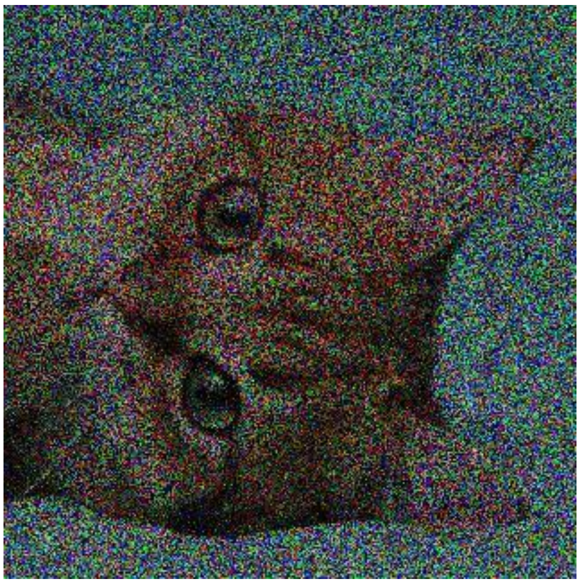


We then applied this trained model to a simulation of image transmission through an AWGN channel for two values of standard deviation sigma. The MATLAB version of the output represents what can be achieved with the current decoding techniques using the dedicated simulation Signal Processing Toolbox.

Sigma = 1 results:

	
Real image	BPSK Neural network output
	
BPSK MATLAB mapping simulation	BPSK python MAP decoding simulation

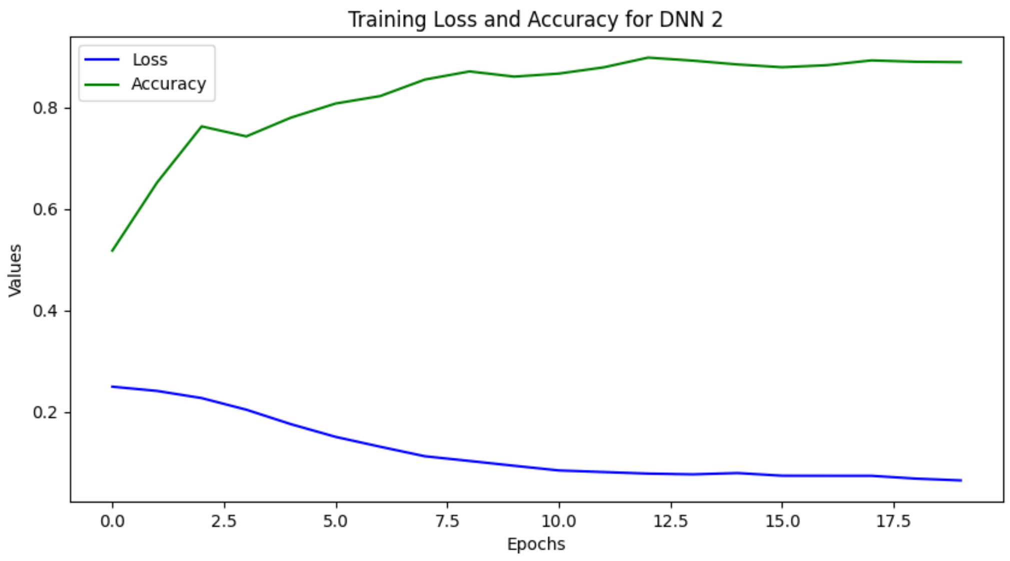
Sigma = 4 results:

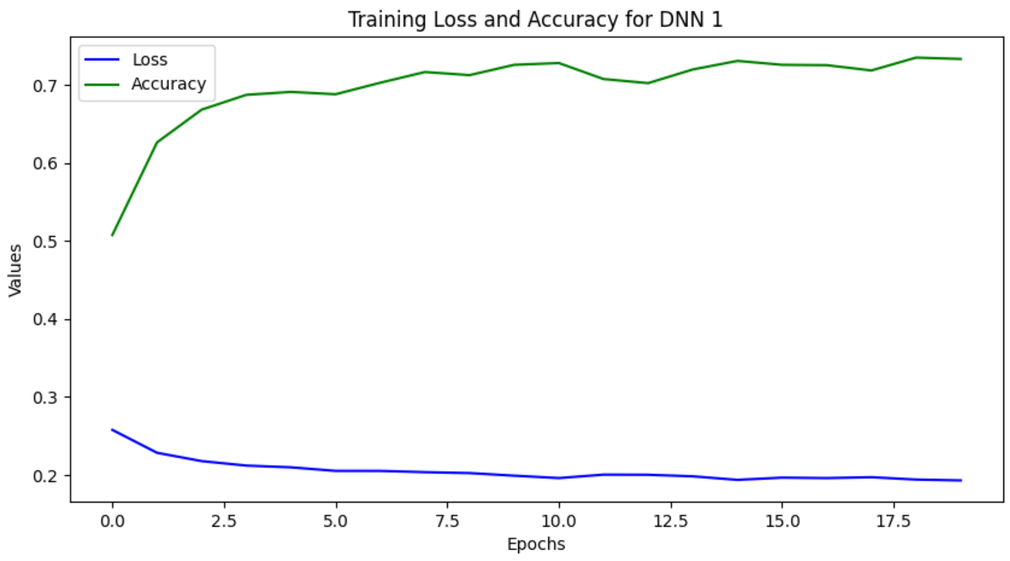
	
Real image	BPSK Neural network output
	
BPSK MATLAB mapping simulation	BPSK python MAP decoding simulation

c) 16QAM results

	LayeredNN	SequentialLayeredNN
Correct message (on 10 000)	7580	8774
Percentage error	24.20%	12.26%
Percentage accuracy	75.8%	87.74%

By comparing these results with those of the first modulation, we observe a decrease in the performance of the two models. This could be due to specific characteristics of the second modulation that make the decoding task more complex. Despite this, the SequentialLayeredNN continues to show superior performance compared to the LayeredNN in this configuration.

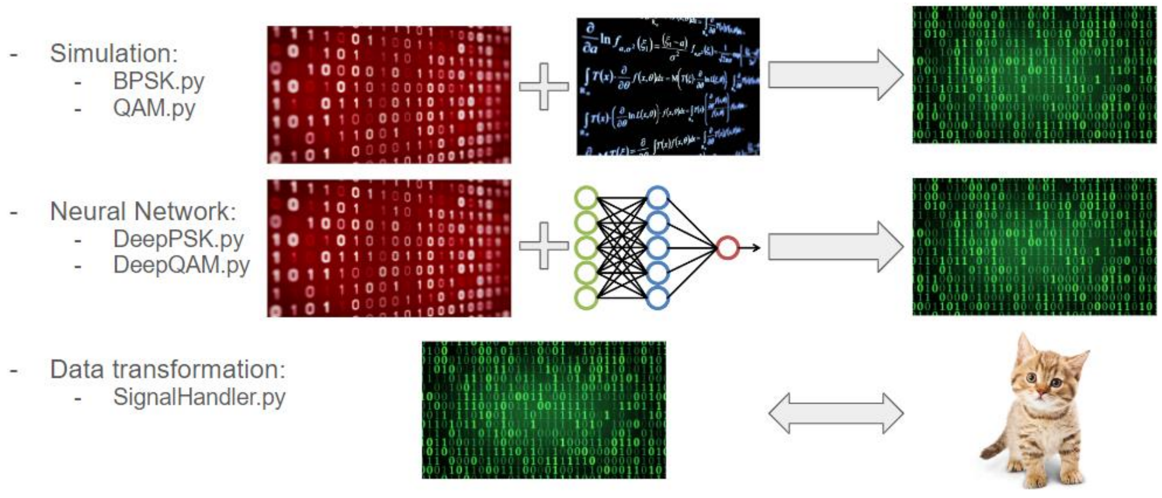




IV- Code structure

a) Operating system

We can divide our code in 3 parts: the data transformation, the simulation part, and the neural network part as shown below:



The simulation related files are used to generate training data, and then to evaluate the decoding performance of different transmission conditions.

The neural network files are training and evaluating neural networks models over simulated datasets.

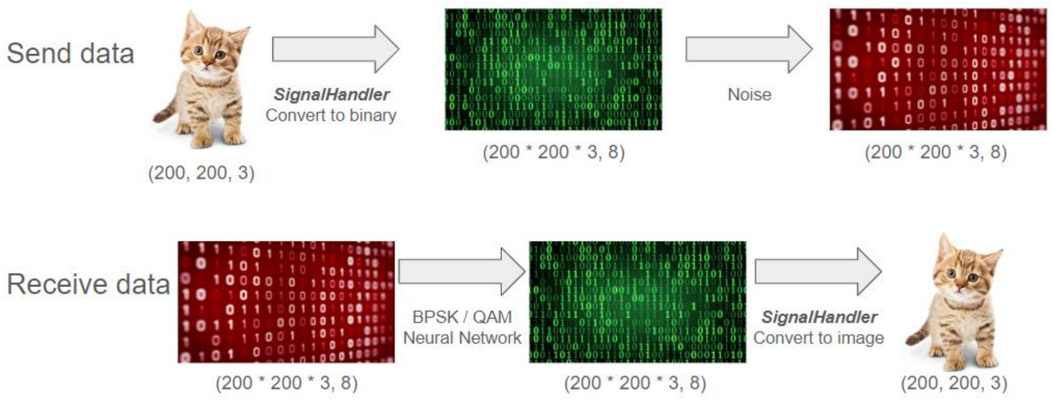
The data transformation file and the main are then called in “real” scenario simulations, such as transmitting the data of an image over a noisy channel.

b) Data transformation method

To send and receive the data we first need to be able to transform a real signal into a binary signal as well as being able to transform a binary signal into a real signal. To do so, we developed a class named ***SignalHandler*** that does these operations.

The class works as follows:

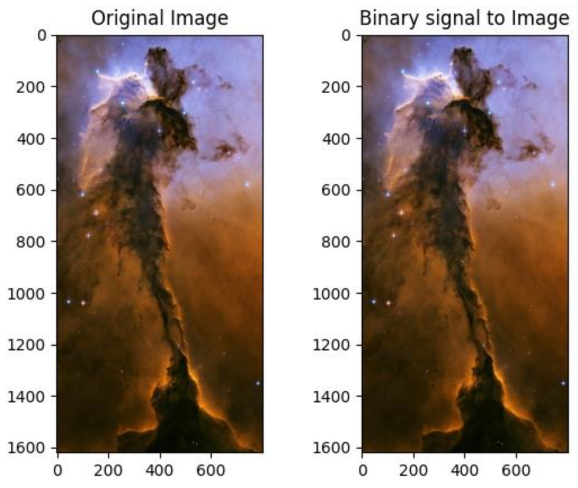
- We can give to the class either an **integer** or a **numpy array** (representing a signal or an image)
- If the input is an integer, we convert the signal into a numpy array containing 1 **row and N columns** (N is the number of bits, in our case N = 8)
- If it is a numpy array we first use the flatten function from numpy to transform the array into a vector, then we transform each value into its binary representation. So, we get a numpy array **M rows and N columns** (N is the number of bits and M the number of values in the array)
- We can also give a binary signal as an input and convert it into the decimal representation. The signal obtained will be a numpy vector of **M rows and 1 column**



Examples of inputs/outputs:

Input	Output
int_sig = 42	int Computation time: 0.00s 42 = [0 0 1 0 1 0 1 0]
bin_sig = [[0, 0, 1, 0, 1, 0, 1, 0], [0, 0, 0, 0, 0, 0, 0, 0]]	binary Computation time: 0.00s Output: [42 0]
img_sig_2 = [[100, 100, 100], [10, 0, 255]], [1, 100, 255], [0, 255, 0]]	array Computation time : 0.00s Output : [0, 1, 1, 0, 0, 1, 0, 0] [0, 1, 1, 0, 0, 1, 0, 0] [0, 1, 1, 0, 0, 1, 0, 0] [0, 0, 0, 0, 1, 0, 1, 0] [0, 0, 0, 0, 0, 0, 0, 0] [1, 1, 1, 1, 1, 1, 1, 1] [0, 0, 0, 0, 0, 0, 0, 1] [0, 1, 1, 0, 0, 1, 0, 0] [1, 1, 1, 1, 1, 1, 1, 1] [0, 0, 0, 0, 0, 0, 0, 0] [1, 1, 1, 1, 1, 1, 1, 1] [0, 0, 0, 0, 0, 0, 0, 0]

To verify the binary to real conversion, we convert an 8-bit image into a binary signal, then we re-convert the binary signal into a real signal, and we effectively obtain the same result, as shown below:

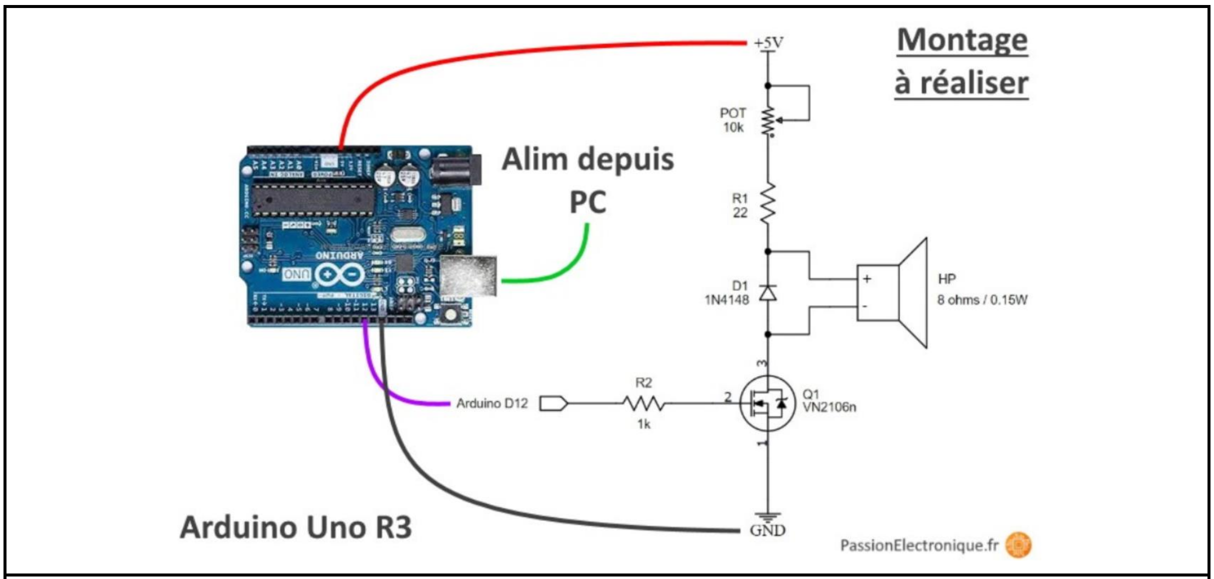


The main limitation of our class is that it remains limited to integer values, it is not possible to handle decimal values. We can nevertheless try to find tricks to compensate for this limitation, such as re-mapping decimal values to integers.

V- Real signal

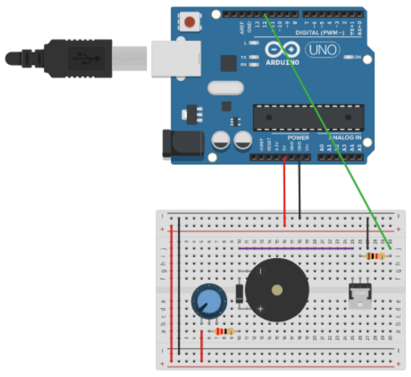
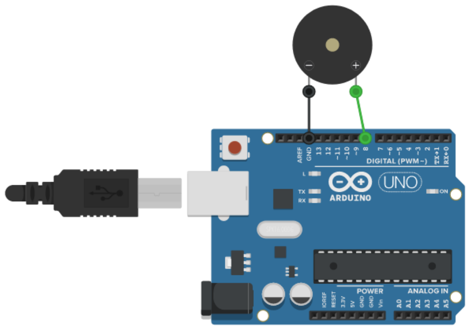
a) Setup

As part of this project, we wanted to make a real-life application to validate our proof of concept. We therefore imagined transmitting a bit frame with sound disturbed by ambient noise. To do this, we are going to use two Arduino boards. One as a transmitter and one as a receiver. Arduino boards are easy to program and will enable us to build a prototype quickly. The physical medium will be sound. The transmitter uses a speaker, while the receiver uses a microphone. The advantage of using sound is that it is very easy to realize how noisy it is, and the rendering will be very meaningful. For the speaker, an impedance-matched assembly is required to avoid damaging the components:

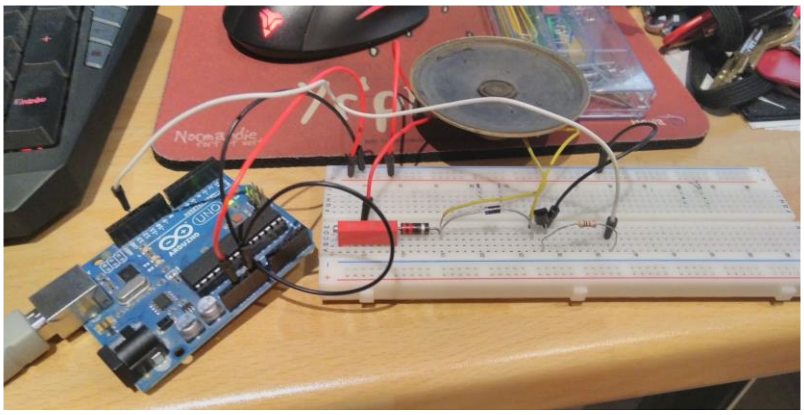


Electrical diagram

The speaker is powered by +5V from the board. It is controlled by a switching transistor protected by a resistor. To control the intensity of sound, a potentiometer is placed to the power supply, along with a protective resistor. A discharge diode is also placed in parallel of the speaker. The schematics are as follows:

	
Transmitter modelled on TinkerCAD	Receiver modelled on TinkerCAD

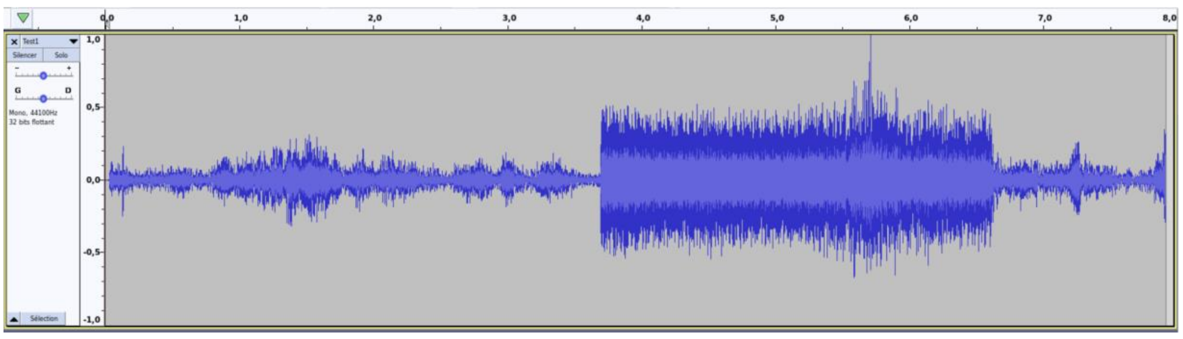
We then set about making the electrical connections. Unfortunately, we did not have the KY-037 transducer (microphone), and time constraints forced us to choose another solution. So, we decided to use a smartphone to record the sound. Only the transmitter part will be provided by an Arduino Uno board. The drawback is that we cannot choose the sampling frequency.


Transmitter set-up

Once the assembly was complete, we transmitted a binary message with the following properties:

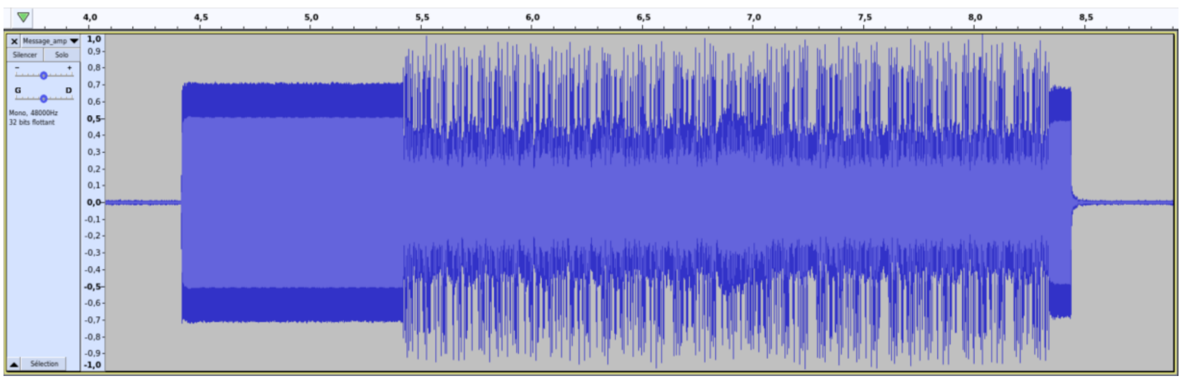
- 400Hz carrier frequency
- 125Hz bitrate
- 8kHz sampling frequency

From there we were able to proceed with a first test.



Signal rendering on Audacity

Unfortunately, we were unable to decode the signal because we could not find the exact moment when it started, and because the background noise was irregular. In order to capture the moment when the message starts, we added a 2 kHz slot signal for one second just before the message and repeated the test under quieter conditions.



Signal rendering on Audacity

b) Results & improvements

Although theoretically working considering the parameters of the real-world simulation, this setup raised an unplanned issue that could not be solved, thus making the experiment invalid.

The slot signal was not enough to precisely decide the starting point of the transmission, as a small-time delay in the sampling would result in a large dephasing, which distorts totally the decision made by the neural network evaluation.

To overcome this issue, the neural networks models should be retrained with this new objective in mind, with the following improvements:

- To learn to decode pattern-based signals rather than a series of values (i.e. to put emphasis on convolution layers)
- To learn over examples of varying length instead of word-based training examples

VII - Conclusion

This study tackled the issue and challenges raised by the noise perturbation in telecommunications. By reproducing similar results to the state-of-the-art hardware, we proved possible the use of deep learning in the context of noise cancelling.

Although the simulated conditions are not identical to the one found in real application, we also proved the possibility of having neural network transfer learning applied to signals with different simulation parameters yet similar wave patterns after encoding.

Thus, the first step of this proof of concept may considered achieved, and opens the door to further improvements, notably in the fields of the neural network architecture, by reinforcing its ability to recognise patterns rather than values, as well as deepening the network, so it may decode more complex signals, such as 4096 QAM ones, used in TV broadcasts.