## Arguments

1. channel: defines the model channel for the noise, possible values 'BAC' and 'BSC'
2. N: any value power of 2
3. k: any value < N
4. Iterations: number of iterations for the iterative fine tuning (should be an even number)
5. Packets: Number of packets used to compute the metrics (BER and BLER)
6. type of training: the possible values are
   - bug: use to know if everything works well, the results are not useful
   - short: for short training, some results are useful but not recommended
   - medium: not a very long training, take some time, best results were obtained using this one
   - long: very long training, some models can overfit and for others longer training doesn't improve the results.

## 1. Array input Array output

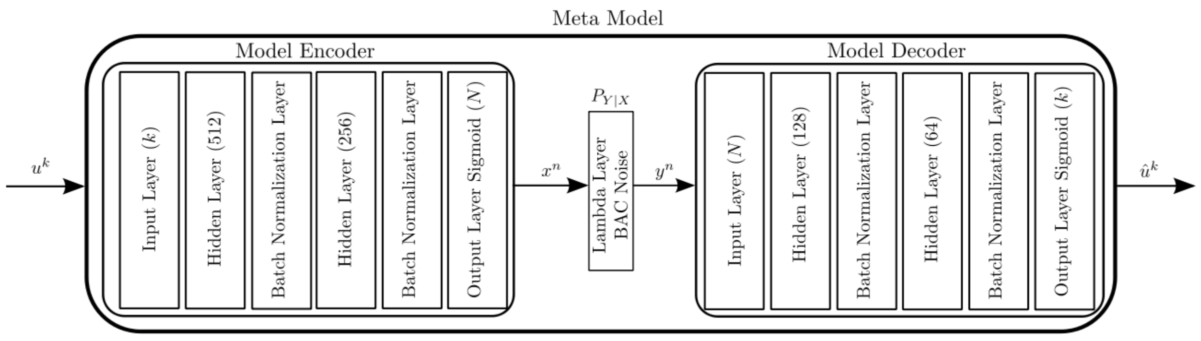
In all those solutions presented below we consider an encoder model composed by 2 layers with activation function MISH, SeLU or ReLU, all those give rise almost to the same result. Always using Batch Normalization with default parameters and as output layer a sigmoid.

For the decoder, similarly, the model is composed by 2 layer with activation function MISH or ReLU + Batch Normalization and output layer sigmoid. In training the input is always the output of the lambda layer that models the noise.

The loss function is the Mean Squared Error, it was tested the categorical cross entropy but it doesn't work well.

And initializer HeNormal is used, because I found in some documentation that this has been proved to be the best for ReLU and sigmoid layers. It would be interesting to test more this element.

All the models presented in this section were built as shown in this figure:



### a) autoencoder_array-array_fine-decoder.py

In this file, we pretrain the encoder and decoder jointly, then we freeze the decoder and train the encoder, then we freeze the encoder an train the decoder, this process is repeated according to the number of iterations.

I use a callback to reduce the learning rate by 0.001 each epoch. Different optimizers were also tested (Adam, SDG, Nadam, etc) with different parameters. Finally, the Nadam with default values give us the best results.

Each stage (pretrain, encoder or decoder train) can have a different epsilon to train. Many different values and combinations were tested and at the end the best results were obtained when the pretrain was 0.03 and keep it all along the training, if the decoder increases a little bit (up to 0.07) the decoder seems more to the MAP for the learned encoder.

How to execute it: \Python3\python.exe autoencoder_array-array_fine-decoder.py *N k iterations packets type*

Example: \Python3\python.exe autoencoder_array-array_fine-decoder.py BAC 8 4 0 1000 medium

### b) autoencoder_array-array_alt-training_loss-validator_regularizer.py

The regularizer used forces the output of the encoder model to be closer to the values 0 and 1, considering than the output is sigmod function.

In this file, we pretrain the encoder and decoder jointly, without regularizer. Then a sort of fine tuning where we freeze the decoder and train the encoder using a linear regularizer, then we freeze the encoder an train the decoder, in this case the regularizer doesn't affect, this process is repeated according to the number of iterations.

Same considerations for the optimizer and callback.

In loss graph we can see a test for multiple values of epsilon, and how the accuracy evolves each epoch.

How to execute it: \Python3\python.exe autoencoder_array-array_alt-training_loss-validator_regularizer.py *N k iterations packets type*

Example: \Python3\python.exe autoencoder_array-array_alt-training_loss-validator_regularizer.py 4 2 2 1000 medium

### c) autoencoder_array-array_alt-training_loss-validator_regularizer_interval.py

The interval is a one-hot vector of length 4, where intervals are regularly distributed in [0, epsilon_0_training], it has be found that 0.1 is a good value for the epsilon_0_training, considering the trade-off between big and small epsilon in terms of BER. 70% data belongs to the first interval, 15% to the second, 10% to the third and 5% to the fourth (consider to look for the optimal repartition).

In this file, we pretrain the encoder and decoder jointly. Then a sort of fine tuning where we freeze the decoder and train the encoder using a linear regularizer, then we freeze the encoder an train the decoder, in this case the regularizer doesn't affect, this process is repeated according to the number of iterations.

Same considerations for the optimizer and callback.

In loss graph we can see a test for multiple values of epsilon, and how the accuracy evolves each epoch.

How to execute it: \Python3\python.exe autoencoder_array-array_alt-training_loss-validator_regularizer_interval.py *N k iterations packets type*

Example: \Python3\python.exe autoencoder_array-array_alt-training_loss-validator_regularizer_interval.py 4 2 2 1000 medium

### d) autoencoder_array-array_polar_alt-training.py

In this file, we pretrain decoder to be the optimal decoder for a polar code, and then an encoder is pretrained to reproduce the polar encoder. After that a sort of fine tuning where we freeze the decoder and train the encoder, then we freeze the encoder an train the decoder. This process is repeated according to the number of iterations. Finally, a jointly fine tuning is made with a very low learning rate.

To obtain good results this model needs more than 4 iterations.

How to execute it: \Python3\python.exe autoencoder_array-array_alt-training_loss-validator_regularizer_interval.py *channel N k iterations packets type*

Example: \Python3\python.exe autoencoder_array-array_polar_alt-training.py BAC 8 4 6 10000 medium

## 2. Array input one-hot output

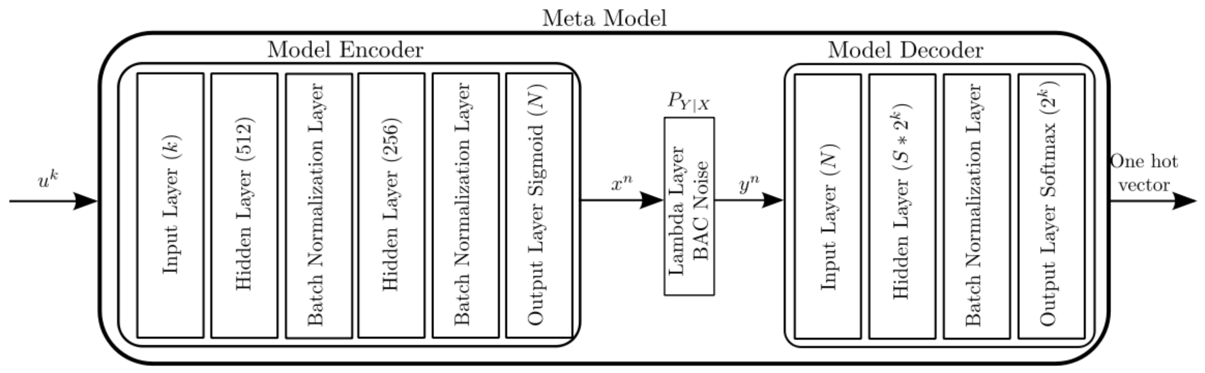
In all those solutions presented below we consider an encoder model composed by 2 layers with activation function MISH, SeLU or ReLU, all those give rise almost to the same result. Always using Batch Normalization with default parameters and as output layer a sigmoid.

For the decoder, the model is composed only by hidden layer with activation function MISH or ReLU + Batch Normalization and output layer softmax. In training the input is always the output of the lambda layer that models the noise.

The loss function is the Categorical Crossentropy, it was tested the mean squared error but it doesn't work well and the KL-divergence, which have same results then crossentropy but slows down the training.

An initializer Ortogonal is used, because I found in some documentation that this has been proved to be the best for ReLU and Softmax layers. It would be interesting to test more this element.

All the models presented in this section were built as shown in this figure:



### a. autoencoder_array-onehot_fine-decoder.py

In this file, we pretrain the encoder and decoder jointly, then in the phase of fine tuning we freeze the encoder and train the decoder. In the code is possible to train the encoder alone, but I notice that is better not to do it, in terms of BER.

Just in pretrain, I use a callback to reduce the learning rate by 0.001 each epoch. Different optimizers were also tested (Adam, SDG, Nadam, etc) with different parameters. Finally, the Nadam with default values give us the best results.

Each stage (pretrain, encoder or decoder train) can have a different epsilon to train. Many different values and combinations were tested and at the end the best results were obtained when the pretrain was 0.1 for pretrain and decoder fine tuning, in the other hand, for encoder fine tuning is better to have a lower epsilon_0_training, around 0.05.

How to execute it: \Python3\python.exe autoencoder_array-onehot_fine-decoder.py *N k packets type*

Example: \Python3\python.exe autoencoder_array-onehot_fine-decoder.py 8 4 1000 medium

### b. autoencoder_array-onehot_fine-decoder_ext-interval.py

In this file, we pretrain the encoder and decoder jointly, with an estimation of the epsilon_0 (interval) as an input of the noise layer and the decoder model. Then a sort of fine tuning where we freeze the encoder and train the decoder. This solution has the best results in general , since for small epsilon_0 values is close to the Polar codes and for big epsilon_0 values is close to the array-array solution.

As always the optimal epsilon_0 for training depends on the results you're looking for, but 0.1 has be found to be a good value. 80% data belongs to the first interval, 10% to the second, 8% to the third and 2% to the fourth (consider to look for the optimal repartition).

Same considerations for the optimizer and callback.

How to execute it: \Python3\python.exe autoencoder_array-onehot_fine-decoder_ext-interval.py *N k packets type*

Example: \Python3\python.exe autoencoder_array-onehot_fine-decoder_ext-interval.py 8 4 1000 medium

### c. autoencoder_array-onehot_fine-decoder_int-interval.py

In this file, we pretrain the encoder and decoder jointly, with an estimation of the epsilon_0 (interval) generated by the noise layer and put as an input of the decoder model. Then a sort of fine tuning where we freeze the encoder and train the decoder.

In the lamba layer that models the noise, the interval is a one-hot vector of length 4, where intervals are regularly distributed in [0, epsilon_0_training], with the same number of elements each one.

How to execute it: \Python3\python.exe autoencoder_array-onehot_fine-decoder_int-interval.py *N k packets type*

Example: \Python3\python.exe autoencoder_array-onehot_fine-decoder_int-interval.py 8 4 1000 medium

### d. autoencoder_array-onehot_fine-decoder_int-interval-irregular.py

In the lamba layer that models the noise, the interval is a one-hot vector of length 4, where intervals are irregularly distributed in [0, epsilon_0_training], with the same number of elements each one. That allows the decoder to be more accurate in the estimation for small values of epsilon_0

How to execute it: \Python3\python.exe autoencoder_array-onehot_fine-decoder_int-interval-irregular.py *N k packets type*

Example: \Python3\python.exe autoencoder_array-onehot_fine-decoder_int-interval-irregular.py 8 4 1000 medium

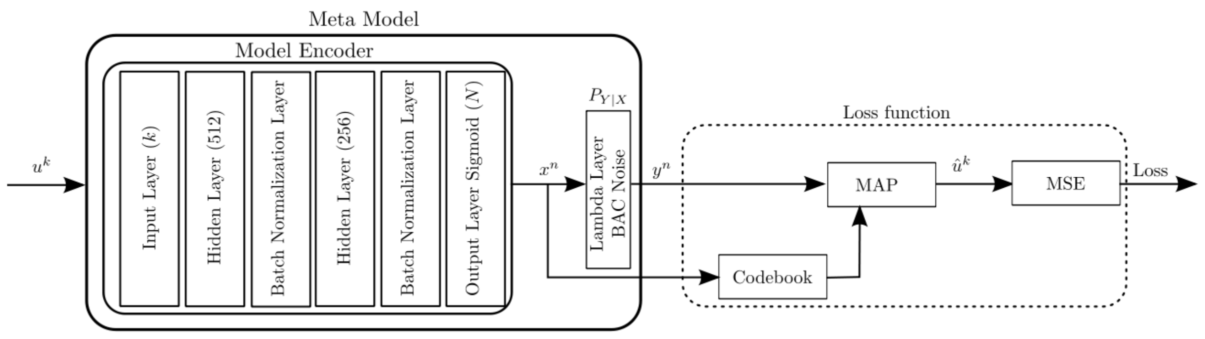
## 3. Array input MAP output

Different solutions were presented to solve problem to find the optimal encoder, are detailed here below

In all those solutions we consider an encoder model composed by 2 layers with activation function MISH + Batch Normalization with default parameters and as output layer a sigmoid.

### a. autoencoder_array-MAP.py

In this file, we pretrain an encoder to reproduce the polar encoder. Then a sort of fine tuning is made over the meta-model using a custom loss function which find an estimate of the transmitted message an then computes the Mean Squared Error as shown in the figure here below.



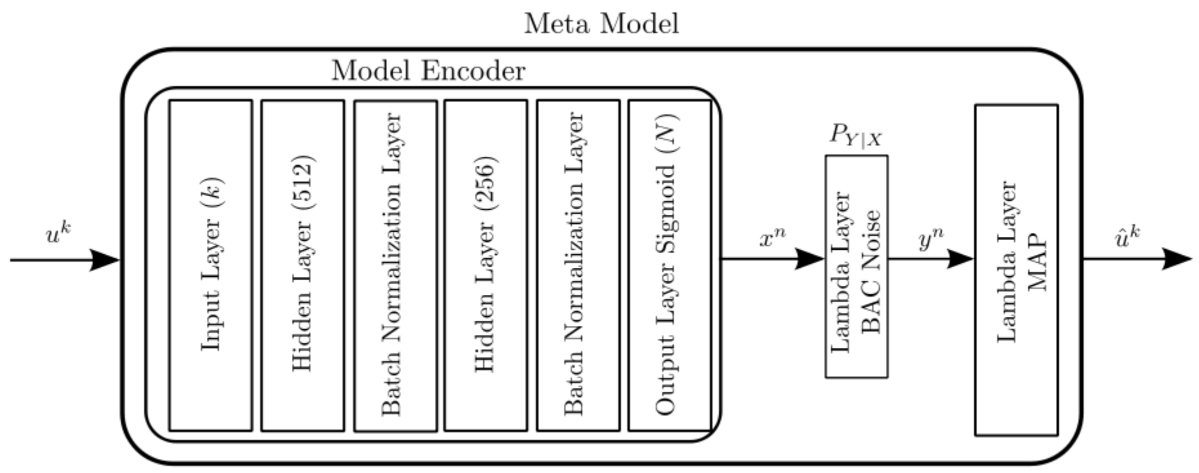
This model has some problems, I think because the model keeps trapped in some local minima.

How to execute it: \Python3\python.exe autoencoder_array-MAP.py *N k packets type*

Example: \Python3\python.exe autoencoder_array-MAP.py 8 4 1000 medium

### b. autoencoder_array-MAP-lambda.py

In this file, we replace the decoder by a lambda layer that computes the estimation for u^k, this system also stays in local minima when training, but obtains some result.



How to execute it: \Python3\python.exe autoencoder_array-MAP-lambda.py *N k packets type*

Example: \Python3\python.exe autoencoder_array-MAP-lambda.py 8 4 1000 short

### c. autoencoder_array-MAP-lambda-soft.py

In this file the architecture is the same than the last one, the change is that the neural network MAP decoder doesn't return the estimation but the probability distribution, it's some kind of softmax. You can probably explore it more, changing for example the epsilon training or solving the rounding problem. I really think this architecture could work well.

How to execute it: \Python3\python.exe autoencoder_array-MAP-lambda-soft.py *N k packets type*

Example: \Python3\python.exe autoencoder_array-MAP-lambda-soft.py 8 4 1000 medium

## Future work

- Do more exhaustive analyses about the optimal epsilon for training; probably finding a theoretical way to do it. That will reduce the implementation and test time.
- Find and define the optimal initializer, that was a thing I don't really take into account. That will reduce the implementation and test time, as well as will give you a better benchmark of comparison between architectures and training techniques.
- Reduce the randomness of the training, setting a custom seed for the initializer and the noise.
- Consider comparing the autoencoders trained for the BAC and the BS, in particular, for small values of epsilon_0, considering that the results I obtained are good more specifically when the channel is more asymmetric.
- Find the good distribution in the amount of elements for each interval, because the distribution I made was an approximation, then try to find a real way to determine it.

- Explore more the effect of regularizers (forcing the encoder output in {0,1}) for me the best was the linear but maybe for other architectures the quadratic could perfect. Another parameter to consider is the weight given to this regularizer. I tried given the same weight than the loss function and works well but It’s probably not the optimal.
- Change the output activation function in the encoder for a linear or a tanh, this will reduce the gradient vanishing problem and probably help to find the real minima and not just a local
- Find a way to give and estimation of the channel to the encoder, something similar what we do with the decoder, and then evaluate the error when a bad estimation is given.
- Explore deeply the solution of training the encoder with a lambda layer that reproduces the MAP, maybe an  activation function different to the sigmoid at the output will improve the results.