ETH zürich

# Technical Report

---

# Spin-orbit torque magnetic tunnel junctions for neuromorphic computing

---

Dashiell Harrison

April 2025 - August 2025

**Internship supervisor :**

Pr. Pietro Gambardella

**Internship supervisor :**

Marco Hoffmann

**Home university :**

ENS | PSL

**Host department :**

D MATL

# Table of contents

# Introduction

This report is a continuation of the ICFP M2 report, which introduced the work carried out during my master's internship in Prof. Gambardella's Magnetism and Interface Physics group. The present document focuses more specifically on the technical aspects and on the results of the experiments. It is organized as follows:

- Part 1 describes the implementation of a reservoir computing scheme using MTJs,
- Part 2 presents the experimental protocols developed for classification and prediction tasks,
- Part 3 discusses the results obtained from the different experiments.

# Part 1 :
# Reservoir computing

## 1.1 Reservoir

Reservoir computing originates in 2001 under the name of "Echo State Network" (ESN) [1]. It is an algorithm that follows the scheme of a recurrent neural network (RNN). A RNN is a neural network scheme where an input can travel through the same neuron several times before it delivers an output.

In the particular case of reservoir computing, the 'recurrent network' is the reservoir which is a fixed network of nonlinear units. Mathematically, a reservoir is a 'nonlinear generalization of standard bases' [2]. The output of the reservoir is the reservoir state, it is not the output of the whole neural network. Indeed, the reservoir state goes through a last neural network layer, called the output layer, which then yields the final result. In reservoir computing, only the output layer of the neural network is trainable to achieve learning.

To resume, reservoir computing is a fixed reservoir together with a trainable output layer. In an ESN, the reservoir is an abstract graph with a fixed topology, i.e. with a fixed density of links between the nodes of the graph. In physical reservoir computing, the reservoir is a fixed physical system composed of many non-linear units. In the present case, one MTJ is the reservoir. The magnetic domains in the MTJ are the units, i.e. "neurons" that interact non-linearly with the input signal to give a switching probability. The switching probability is the reservoir state. Finally, the switching probability goes through the output layer, which is typically a linear or ridge regression [3]. The weights of the neurons in the output layer, symbolized by the matrix $W^{\text{out}}$ in 1.1, are then adjusted during the training process to achieve specific classification or prediction tasks (see 2).

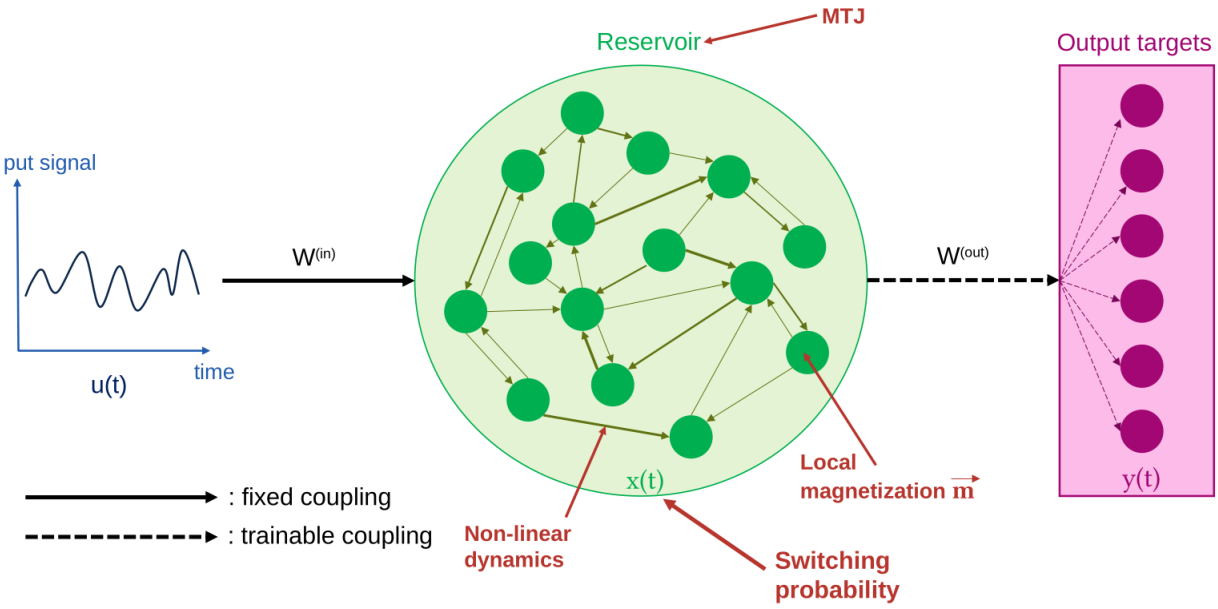



Figure 1.1: Reservoir computer scheme with one MTJ

NB : adding more MTJs to the reservoir widens the reservoir. At some point, we could consider each MTJ as an individual neuron. This would even better harness the sigmodal switching probability of the MTJ which has the same shape as the activation function of one neuron: one MTJ would then be one neuron in a reservoir computer architecture. We artificially implement this idea in the prediction tasks methods (see 2.3) or in some classification attempts (see 3).

## 1.2 Input layer

To input the initial data in the reservoir, one can either input the raw data or filter it with input matrices before going through the reservoir. In the literature ([4], [5], [6]), it is prefered to use large sparse matrices as input layer for computational efficiency. However, applying a filter will depend on the encoding scheme (see 2.2.2) and the reservoir itself. Indeed, if the initial data is simple enough to be processed by the reservoir, there is no need to add a filtering input matrix. In our case, if the raw data is composed of too many voltage pulses with values that would almost systematically switch the MTJ, then these values have to be tampered down. Otherwise, all of the data will show a 100% switching probability and none of it will be distinguishable. But if the raw data is already sparse enough, meaning that some will switch the MTJ and some won't, then there is no need to apply an input matrix different from the identity.

## 1.3 Output layer

The switching probability is then mapped back onto a training target through the output layer. The training targets are the results of our neural network. If we are performing handwritten digits computer vision classification tasks, then the target vectors

are one-hot vectors (see 1.3.1) representing the handwritten digit to be recognized. If we are performing time series forecasting, then the target vectors are data points (x,y,z coordinates) that come temporally after the input data points.
The output layer takes the shape of a matrix : $W^{\text{out}}$. It is the only trained layer. It is obtained by minimizing a loss function. The study used the following [2]:

$$W^{\text{out}^T} = \min_w(||\mathcal{P}_{sw}w - y||^2 + \beta||w||^2) \tag{1.1}$$

where :

- $\mathcal{P}_{sw}$ is the switching probability
- $y$ is the $1\times10$ target vector with 0 everywhere except at the digit position (one-hot vector)
- $\beta \in [0, 1]$ is a regularization parameter

For $\beta = 0$, minimizing the above loss function amounts to computing a pseudo-inverse matrix. It is hence a linear regression or 'ordinary least squares' method. With $\beta \neq 0$, minimizing the above loss function is called a ridge regression.

To map a switching probability vector to a target vector, $W^{\text{out}}$ has to have the shape of the switching probability vector dimension times the target vector dimension.

### 1.3.1 Example

If one tries to classify the MNIST handwritten digits 8 and 3, the target vector dimension is 2 because there are 2 classes: $\begin{bmatrix}0\\1\end{bmatrix}$ and $\begin{bmatrix}1\\0\end{bmatrix}$. To classify these digits, one can decide to send each digit pixel-sequence at once in the reservoir and attribute a scalar switching probability to each digit. This means that the dimension of $W^{\text{out}^T}$ is $2 \times 1$. For example :

$$W^{\text{out}^T} = \begin{bmatrix}0.5\\1.\end{bmatrix} \tag{1.2}$$

These are the linear coefficients of the regression. But the model we use also computes the intercepts : there are as many intercepts as there are classes, here there are 2 :

$$W^{\text{out}^T}_{intercepts} = \begin{bmatrix}0.3\\0.02\end{bmatrix} \tag{1.3}$$

In the testing phase, a switching probability equal to 0.8 in this example will be classified by the model as an 8 digit:

$$\begin{bmatrix}0.5\\1.\end{bmatrix} \times 0.8 \ + \ \begin{bmatrix}0.3\\0.02\end{bmatrix} = \begin{bmatrix}0.7\\0.82\end{bmatrix} == \begin{bmatrix}0\\1\end{bmatrix} \tag{1.4}$$

The == shows how the regression classifies the result : the highest value of the matrix is set to 1 and all the others to 0. With a switching probability of 0.1, the model would predict a handwritten 3 :

$$\begin{bmatrix}0.5\\1.\end{bmatrix} \times 0.1 \ + \ \begin{bmatrix}0.3\\0.02\end{bmatrix} = \begin{bmatrix}0.35\\0.12\end{bmatrix} == \begin{bmatrix}1\\0\end{bmatrix} \tag{1.5}$$

The regression will work only if the reservoir projects the initial objects in different parts of the output mathematical space : $[0,1]^n$ where $n$ is the dimension of $\mathcal{P}_{sw}$. It comes naturally that a higher-dimensional switching probability will facilitate the regression for more classes. This is called adding features to the dataset. Cutting the pixel sequence of a handwritten digit into 3 and getting a switching probability for each of them will add 3 features to the $\mathcal{P}_{sw}$ before undergoing the linear or ridge regression.
In the case of 3 switching probabilities per object and 10 classes (handwritten digits from 0 to 9) then ${W^{\text{out}}}^T$ would have dimensions $10 \times 3$ and ${W^{\text{out}}}^T_{intercepts}$ would have dimensions $10 \times 1$. For example:

$$ {W^{\text{out}}}^T = \begin{bmatrix} 10. & 0.2 & 0. \\ 1. & 0.7 & 19. \\ 0.8 & 25. & 10. \\ 25. & 7. & 23. \\ 35. & 0.1 & 0.2 \\ 37. & 1.2 & 1.3 \\ 42. & 9. & 10. \\ 8. & 24. & 0.3 \\ 0.9 & 0.4 & 1.7 \\ 10. & 10. & 5. \end{bmatrix} \tag{1.6} $$

and

$$ {W^{\text{out}}}^T_{intercepts} = \begin{bmatrix} 1.0 \\ 0.2 \\ 0.02 \\ 0.1 \\ 0.3 \\ 5. \\ 7. \\ 0.8 \\ 7. \\ 0.1 \end{bmatrix} \tag{1.7} $$

applied to the following 3 switching probabilities obtained by sending through the MTJ the pixel sequence representing a handwritten digit in 3 parts:

$$ \begin{bmatrix} 0.3 \\ 0.4 \\ 1.0 \end{bmatrix} \tag{1.8} $$

would yield :

$$\begin{bmatrix} 4.08 \\ 19.78 \\ 20.26 \\ 33.4 \\ 11.04 \\ 17.88 \\ 33.2 \\ 13.1 \\ 9.13 \\ 12.1 \end{bmatrix} == \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{1.9}$$

Classified as a handwritten 6.

## 1.4 Scoring

To test whether the neural network is well trained, different quantities can be computed after sending a test data set through it. For classification, the relevant quantities can be found in a confusion matrix (see figure 1.2).

- True positives (TP): for a given class, the number on the diagonal is the number of correctly identified digits (hence the number of true positives).
- True negatives (TN): for a given class, the number of data points that were correctly classified in another class (rest of the diagonal).
- False positives (FP): for a given class, the number of data points that are in the row of the output class but not in the target class (falsely identified as the class).
- False negatives (FN): for a given class, the number of data points that are in the column of the target class but not in the output class (incorrectly classified as another class).

From these quantities, many scores can be computed to assess the strength of the model (accuracy, precision, recall, F1). We mainly chose the accuracy:

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN} \tag{1.10}$$

Moreover, all reported accuracies are obtained using a 5-fold **cross-validation** procedure. In this approach, the dataset is randomly partitioned into five disjoint subsets of equal size. At each iteration, four subsets are used for training while the remaining one is reserved for testing. The process is repeated five times, such that every subset serves once as the test set. The accuracy is computed for each fold, and the final reported value corresponds to the average accuracy across all five folds. This procedure reduces the variance associated with a single train–test split and provides a more reliable estimate of the model performance.

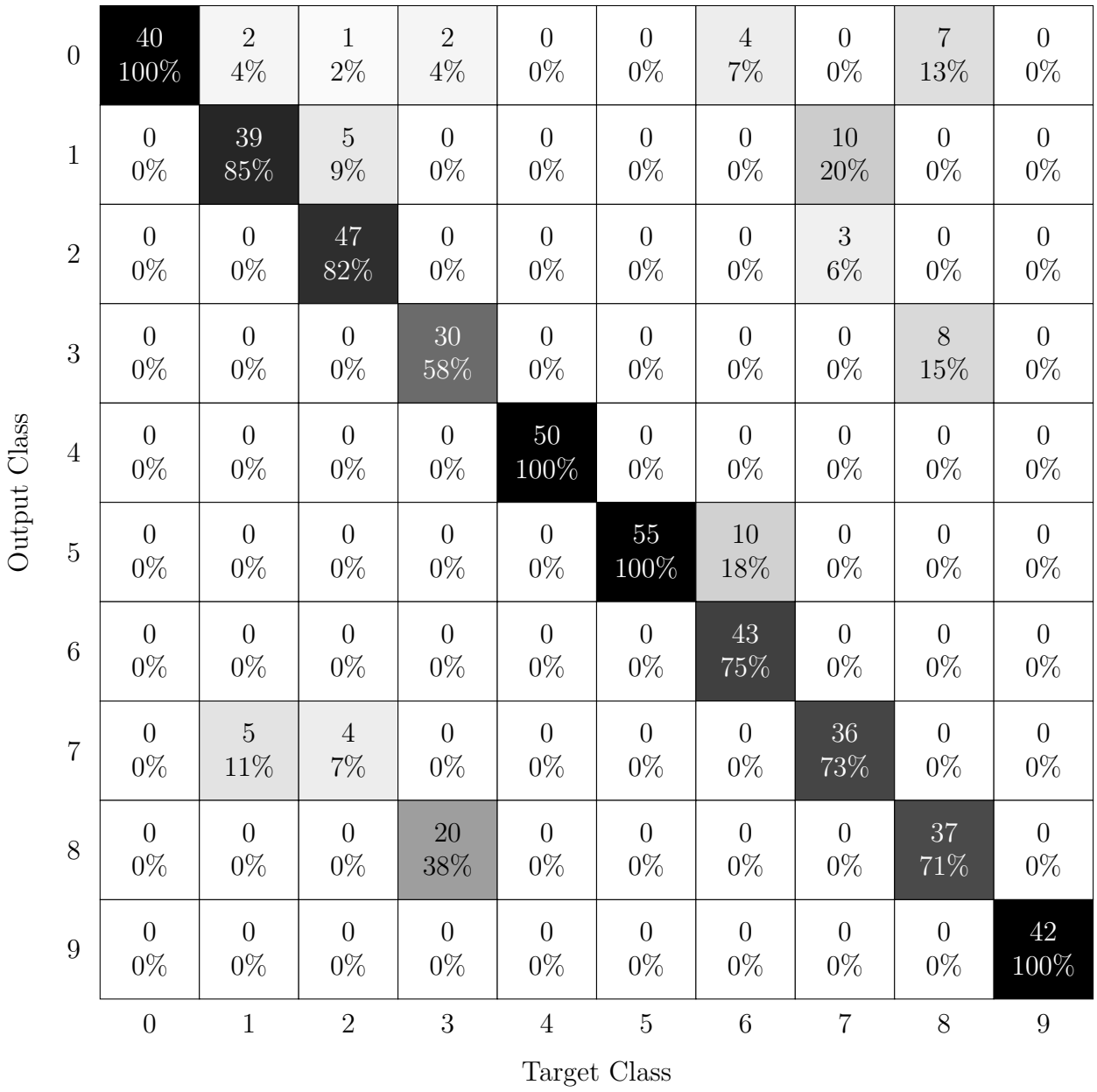



Figure 1.2: Confusion matrix example with a total of 500 testing data points, each classified in one of ten classes: the handwritten digits from 0 to 9.

For prediction tasks, the test score is computed by comparing the value of the predicted data point with the actual one. For example, if one wants to predict the number 4 in the following sequence : [0, 1, 2, 3, 4], then one will feed to the reservoir the sequence [0, 1, 2, 3] and ask the model to predict the next data point, which could be [3.8] for example. The score of the test will compare 3.8 and 4. Among many scores that could be computed (mean square error, mean absolute percentage error, max error, ...), we mainly chose the root mean square error (RMSE):

$$\mathrm{RMSE}(y, \hat{y}) = \sqrt{\frac{\sum_{i=0}^{N-1}(y_i - \hat{y}_i)^2}{N}} \tag{1.11}$$

To evaluate reservoir computing performance, the reservoir can be removed and the accuracy can be compared with the linear model alone. If the dataset is linearly dispersed in the first place, the reservoir is useless. The added value of the reservoir is to transform non-linear data into linearly separated data that can be sorted out with a linear regression. When the comparison is made, the same input layer is kept such that only the reservoir is evaluated. We will see that for binary classification, the output layer alone is sometimes enough to have 100% accuracy and performs well (over 90%) otherwise. However, it performs badly for time series forecasting and for multinomial classification, this is where the reservoir becomes interesting.

One can also compare the neural network to a traditional echo state network and see how many algorithmic nodes it takes to get the same result as the physical reservoir. This has not been implemented here, but one can retrieve an echo state network Python code from the internet and compare the reservoir results to it.

# Part 2 : Methods

## 2.1 Experimental setup and code

The setup (figure 2.1) starts with a 25 GHz-bandwith arbitrary waveform generator (AWG) from which two pulses $V_{SOT}$ and $V_{STT}$ are sent with no relative delay. The pulses can be observed on a 10 GHz-bandwith scope (see figure 2.2) thanks to pick-off tees. The setup continues with bias-tees which enable resistance measurements through the 10 Hz PXI (almost direct current) and the 1 kΩ reference resistance (of the same order of magnitude as the MTJ's resistance, cf. ICFP report).
To prevent the insulator layer from burning when applying excessive voltage, we impose $V_{STT} = 0$ V. To do so, since the MTJ is located at the middle of the SOT track and $V_{out} = 0$ V at the end of the track, we need to apply :

$$V_{STT} = \frac{V_{SOT}}{2} \tag{2.1}$$

in other words $x = 0.5$.

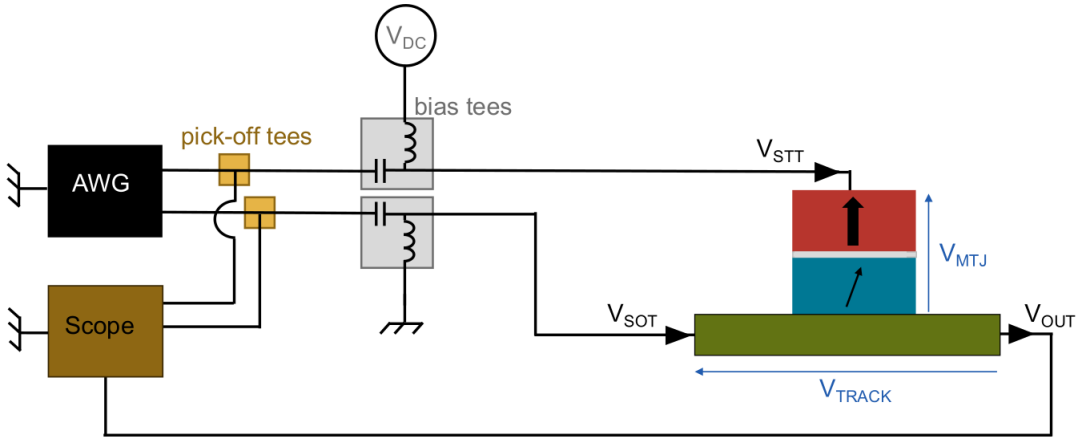



Figure 2.1: Experimental setup diagram. When $\frac{V_{STT}}{V_{SOT}} = 0.5$ we have pure SOT switching.

Pure SOT switching means that we do not take advantage of any STT or VCMA effect that would inhibit or promote switching. However, these effects can be implemented by changing the value of the ratio $x$.

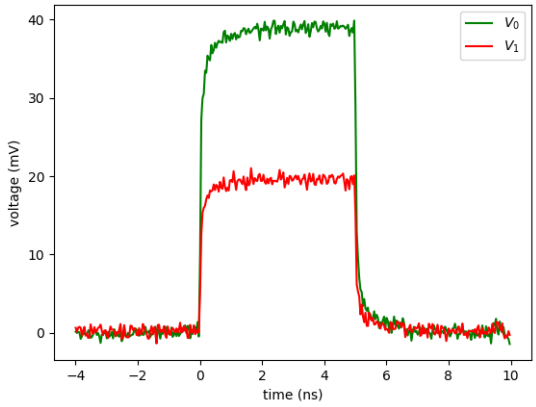



Figure 2.2: Example of 5 ns pulse sent from the AWG and read on the oscilloscope with $V_0 = 0.45\ V$ and $x = 0.5$.

After the pulse, measuring the resistance of the MTJ with the PXI gives access to the state (either P or AP) of the MTJ.

## 2.2 Protocol for classification tasks

This section describes the database, encoding for the input layer, and training of the output layer for computer vision tasks. In this report, the computer vision task is the classification of handwritten digits.

### 2.2.1 Handwritten digits - Scikit-learn database

The MNIST database of handwritten digits is a common benchmark for computer vision tasks (see [7] and [8]). In our first efforts, however, we used the simpler version from the Scikit-learn library in Python. It is composed of 1797 total digits ranging from 0 to 9 each represented by a 64-pixel grayscale array (see figure 2.4). These digits are smaller than the 784 pixels per digit of the actual MNIST, but the methods exposed here are transposable to the MNIST database.
Moreover, for the very first experiment, an even simpler database was created : horizontal and vertical bars inscribed in a 2x2 array.

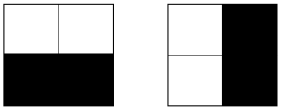


Figure 2.3: First experiment database classes.

### 2.2.2 Encoding

Each grayscale array is converted into a spike train through a type of rate coding scheme [9].
First, a quantification step $t_q$ is defined. Each pixel of the handwritten image represents a pulse of duration $t_q$.
Second, each pixel digital value is treated as a voltage analog value. The pixel array becomes effectively a $t_q$-pulse sequence with each voltage corresponding to the pixel intensity.
Either the sequence is applied to the MTJ as is, or it goes through a preprocessing filter before. As said earlier, sending an entire array as a pulse sequence might invariably switch the MTJ and prevent it from correctly classifying the different images. The filters tried were :

- "averaging" filter averages every 8 pixels into 1 pixel intensity to have a shorter pulse
- "sparse" filter picks out randomly a certain percentage, e.g. 60 %, of the pixel values to make a sequence of the same length but with only 60 % non-zero values. This is done by multiplying a pixel array of dimensions $1 \times n$ by a $n \times n$ sparse identity matrix with density $d = 0.6$.
- "long" random filter which multiplies a pixel array of dimensions $1 \times n$ by a $n \times l$ matrix of random values between -1 and 1 where $l$ is a relatively big number like 100.
- "short" random filter which multiplies a pixel array of dimensions $1 \times n$ by a $n \times s$ matrix of random values between 0 and 1 where $s$ is a relatively small number like 8.

One can also cut the pixel sequence into smaller ones and get a switching probability for each of them. This would yield a switching probability vector with dimension $d > 1$ associated to each digit (one-hot vector, see 1.3.1). This technique was widely used for the MNIST database (see 2.2.5).

### 2.2.3 Training

The spike train is then sent $\mathcal{N}$ number of times from the AWG to the MTJ. Each time the MTJ switches, a reset pulse is sent to reset the MTJ state. After the $\mathcal{N}$ attempts, the MTJ will have switched $N_{sw}$ many times. Thus, the MTJ's switching probability $\mathcal{P}_{sw}$ associated with this spike train writes :

$$\mathcal{P}_{sw} = \frac{N_{sw}}{\mathcal{N}} \tag{2.2}$$

The switching probability is the final state of the reservoir for each spike train.
The $W^{\text{out}}$ is then trained by minimizing the loss function 1.1. This is rigorously done by the scikit-learn modules Linear Regression (for $\beta = 0$) and Ridge Regression (for $0 < \beta < 1$). If the switching probability becomes a $d >$ 1-dimensional vector these modules still work perfectly.

### 2.2.4 Example

The number 3 from the scikit-learn library can be plotted as a 8×8 grayscale pixels image.

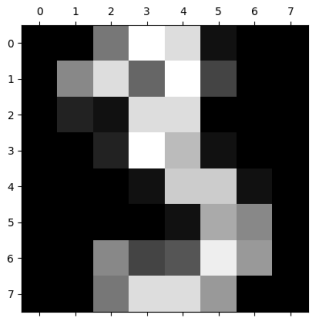
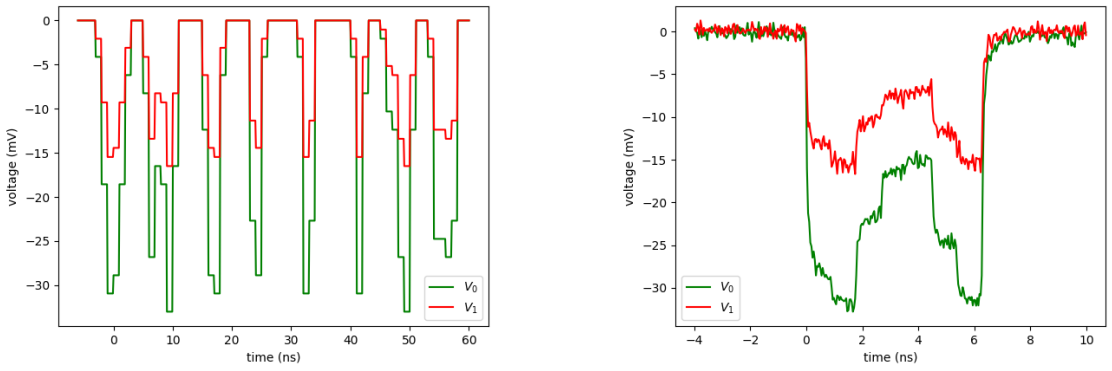


Figure 2.4: Number 3 from scikit-learn handwritten digits dataset.

In Python, it will be represented as a 1×64 matrix, each matrix value carries the grayscale value of the image.

$$\begin{bmatrix} 0. & 2. & 9. & 15. & 14. & 9. & 3. & 0. & \cdots \end{bmatrix} \tag{2.3}$$

The matrix is then normalized and multiplied by the maximum pulse voltage. Henceforth, it becomes a $64 \times t_q$ voltage sequence.




(a) (b)

Figure 2.5: (a) theoretical full pulse of the handwritten 3 and (b) "average"-filter preprocessed signal read on the oscilloscope and sent to the MTJ.
$V_0 = -0.36$ V, $t_q = 0.9$ ns and $x = 0.5$.

To leverage the MTJ's nonlinearity, the pulses go through one of the aforementioned preprocessing filters before being applied to the MTJ.

To add features to the dataset, as mentioned earlier ( 1.3.1, 2.2.2), one can also decompose the pixel sequence into smaller ones. Each 1×64 matrix is decomposed into eight 1× 8 matrices, all of which produce a switching probability. This assigns 8 scalars to each handwritten digit, and the reservoir state is now an 1×8 vector of switching probabilities.

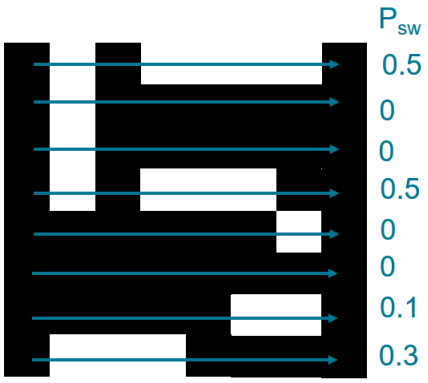



Figure 2.6: Schematic example of assigning 8 switching probability values to a digit 5 of the database.

The switching probability vector is then fed to the output layer as shown in 1.3.1.

### 2.2.5 MNIST handwritten digits database

The reservoir was also trained and tested with the MNIST database. It is composed of 60000 training images and 10000 testing images. One image of a handwritten digit is an array of 28x28 pixels. It is flattened into a $1 \times 784$ such that every valid technique for the

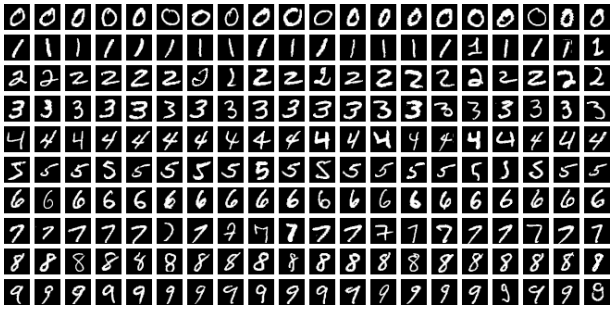


Figure 2.7: Sample of the MNIST dataset

scikit-learn database can be used for the MNIST database as well.
The higher number of pixels makes it harder to harness the nonlinearity of the MTJ because applying 784 positive voltage values to the device will most probably switch it at every try. One can try several solutions to get different switching probabilities according to each digit :

- Artificially add features by cutting the array into a predetermined number of pixel sequences, for example 28 sequences of 28 pixels, and get 28 switching probability values for each digit.
- Apply a filter to the image, for example applying the short random filter will result in a $1 \times 8$ sequence.
- Add an offset to all pixel values to allow reverse switching of the MTJ.
- A mix of all of the above.

For a functional reservoir, one must find a balance between computational efficiency and performance quality of the preprocessing techniques.

## 2.3 Protocol for prediction tasks

This section describes the database, encoding for the input layer, and training of the output layer for time series prediction tasks.

### 2.3.1 Encoding

A time series can be considered as a list of numbers, a simple one to forecast is a repetition the period 01 like a bit stream [10]:

$$\mathbf{01}0101010101010101010101 \ldots \tag{2.4}$$

Each value (i.e. data point) of the time series is interpreted as a voltage value lasting for a time $t_q$ resulting in a pulse. The simplest way of feeding the time series to the reservoir is to send one pulse after another and getting a switching probability for each pulse. This procedure can be done with a simple square wage generator. Another way is to feed of the time series by sequences (e.g. [0101]) and to get a switching probability per sequence. This procedure requires an AWG but does not yield the best results in our first experiments.
Indepently, one can add an offset to the values of the time series. The negative voltage values would not switch the MTJ in a chosen configuration and would add some distinctions to the time series pattern.

More complicated one-dimensional time series exist (periodic with longer periods, aperiodic, Mackey-Glass equation...) and are used as benchmarks in reservoir computing [11]. More over, there also exist $d > 1$-dimensional time series. For example, the Lorenz attractor is in 3 dimensions. In that case, one pulse is composed of 3 pulses of voltage values equal to the $x$, $y$, and $z$ components of each data point. This procedure, although data point per data point, would also justify the use of the AWG.

### 2.3.2 Training

Once an encoding method is chosen, the output layer must predict the next data point by establishing a correlation between each data point. Each data point (or sequence) gives a switching probability, and this switching probability must be sufficient to predict the next data point.
To associate each switching probability with the next data point, the output layer pairs the target vector with every subsequent data point with the preceding switching probability of the $\mathcal{P}_{sw}$ vector.
To do so, the $\mathcal{P}_{sw}$ vector is offset by one 0.0 which is then cut-out (see figure). The green circle is the bit to predict in the 01 bit stream of this example. The training stops right before this green circle. Feeding the last switching probability to the output layer (regression) finally yields the predicted bit.

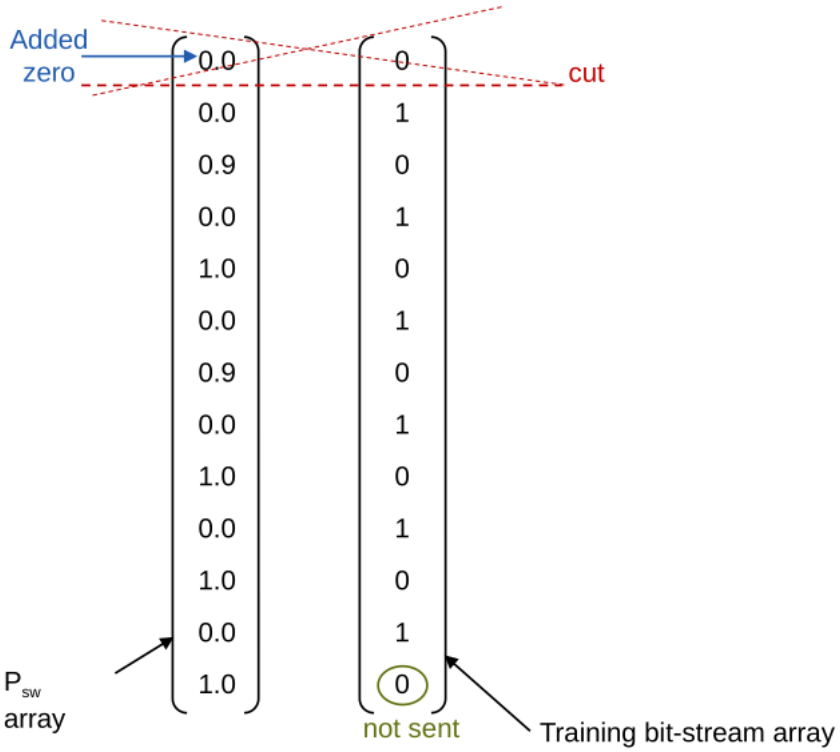



Figure 2.8: Training with the preceding switching probabilities paired to the subsequent bit.

In this example, no memory is needed as there are only two bits, one that switches the MTJ and the other one that does not.

For a more complicated time series, the switching probability needs to carry the memory of the last data points to predict the next one. In order to do so, the switching probability becomes a vector rather than a scalar. This is exactly the same thing as adding features to the dataset. The code produced in this project allows its user to tailor an MTJ architecture presented in figure 2.9. Each data point is transformed as a pulse (either bit-by-bit or sequence-by-sequence). The pulse is then multiplied by a factor (in bold in

the figure) and added to the same pulse multiplied by the previous switching probability (numbers not in bold) and by another factor (in bold again). The sum of these pulses go through the MTJ and yield a new switching probability. The operator has the control over the bold numbers only. For example, with those chosen in figure 2.9:

- the first MTJ from the left yields the same result as what was done until now : the pulse goes through the MTJ untouched. The multiplicative coefficients are $(1, 0)$.
- the second MTJ gets the pulse multiplied by the last $\mathcal{P}_{sw}$. The multiplicative coefficients are $(0, 1)$.
- the third MTJ gets a mixture of both. The multiplicative coefficients are $(0.4, 0.5)$.

This creates a new 3-dimensional $\mathcal{P}_{sw}$ vector to be passed on to the next data point pulse. The initialization is still [0, 0, 0] which is then cutoff as before.

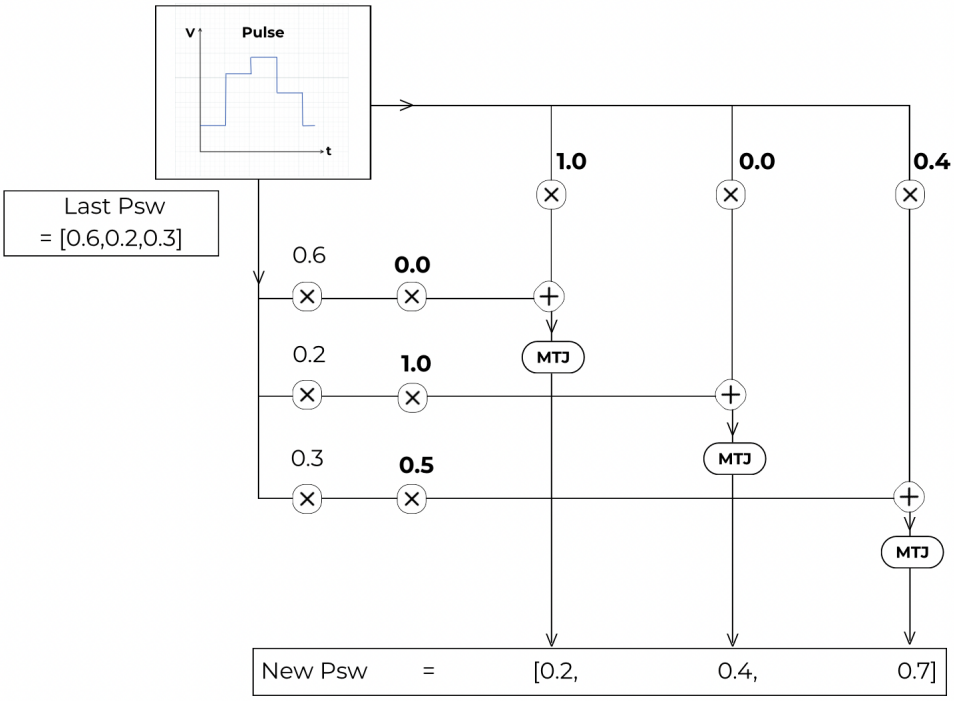



Figure 2.9: Architecture of the reservoir for prediction tasks.

With this procedure, we are printing the memory of the past data points through different channels of MTJ. This architecture showed promising results for simple tasks (see 3). To predict more than one data point, one can inductively predict the subsequent ones by inserting the predicted data point as a pulse in the architecture again.
In this setup, what is actually happening is that we are changing the $x$ ratio. Indeed, with no amplifiers, the max voltage that can go out of the AWG is 0.5 V. When $V_{SOT}$ is already at 0.5 V and $x$ is set at 0.5 then the multiplication by some factor $F$ only abstractly multiplies $V_{SOT}$ by $F$ but what goes out of the SOT channel is still $V_{SOT} = 0.5$ V. However, what goes out of the STT channel is $V_{STT} = x \times F \times 0.5$ V which effectively changes the $x$ ratio by the $F$ factor. Physically, the switching is now assisted by STT

[12]. This method can be burn the junction when the MTJ is fragile. In the present case, only 500 nm wide junctions were tested and all held up even with $x = 1$.

NB : Physically, there is only one MTJ. The operations are done one after the other.

## 2.4 Hyperparameters optimization

The switching probability of the MTJ depends on $V_{SOT}$, $t_q$, $B_x$, and the ratio $x$. These parameters act as the hyperparameters of the reservoir. To identify the combination that yields the highest accuracy, one can employ a grid search algorithm. Various approaches exist [13]. A brute-force grid search tests all possible combinations, but it is time-consuming. An alternative is the tree-structured Parzen estimator (TPE) [14], which is the method used in this report. The "tree structure" in the TPE name refers to the fact that hyperparameters in the search space can be dependent on one another. The Parzen estimator is a Bayesian optimization method, which uses conditional probabilities to divide all hyperparameter values into "good" and "bad" sets (terminology adopted from the research literature) and selects the next combination of hyperparameters accordingly. In practice, it is implemented by the open-source hyperparameter optimization framework Optuna in Python.
More specifically, the TPE begins by testing random combinations of hyperparameters. Then, for each hyperparameter, it fits one Gaussian mixture model $l(p)$ to the values associated with the highest accuracies, i.e. above a certain threshold (by default set at 90% of the best value across all trials): this is the "good" set. Another Gaussian mixture model $g(p)$ is fit to the values that yielded lower accuracies, below this threshold: this is the "bad" set. In subsequent trials, the TPE selects the hyperparameter value $x$ that maximizes the ratio $\frac{l(p)}{g(p)}$. This approach is particularly effective when categorical hyperparameters are involved, as in our case with a filter type (see 2.2.2); this motivated its use here. Nevertheless, other hyperparameter search methods provided by the Optuna framework could also be explored.

# Part 3 : Results

The results are organized into 3 parts :

- classification attempts with 80 nm wide junctions
- classification attempts with 500 nm wide junctions
- prediction attempts with 500 nm wide junctions

## 3.1 Classification with 80 nm wide junctions.

In all of this section, the accuracy given by the ridge or the linear regression was the same. Switching attempts are from P to AP with a positive magnetic field, hence negative pulse voltages. The database is the Scikit-learn one.

### 3.1.1 Binary classification

#### 3.1.1.a First experiment

The first experiment data set consisted of 100 samples of horizontal and vertical bars with a test ratio of 20%. The switching probabilities were determined with $\mathcal{N} = 100$ attempts. The values of the field, the voltage and the quantification step were all tuned by hand. The goal was to separate the vertical from the horizontal bar on the $\mathcal{P}_{sw}$ scale. The best results corresponded to a sub-ns quantification step at low magnetic field and high voltage. This takes advantage of the heating timescale of the free layer (FL) : the vertical bar doesn't heat the FL enough to make it consistently switch while the horizontal bar corresponds to a super-ns pulse which heats the FL enough to switch its magnetization. The separation between the two classes is so clear that one can observe it with his bare eyes (see figure 3.1 and 3.2). Unsurprisingly, the accuracy score is therefore 100 % .

These preliminary results show that a single MTJ can classify two simple shapes by projecting each on a different value range between 0 and 1. This only indicates that the MTJ is a good representation of these shapes but does not give information on the learning capability of the whole system. Moreover, the linear or ridge regression also yields 100% accuracy on this database : the reservoir is useless. We can put it to the test with a harder binary classification task : classifying two numbers of the scikit-learn digits database.

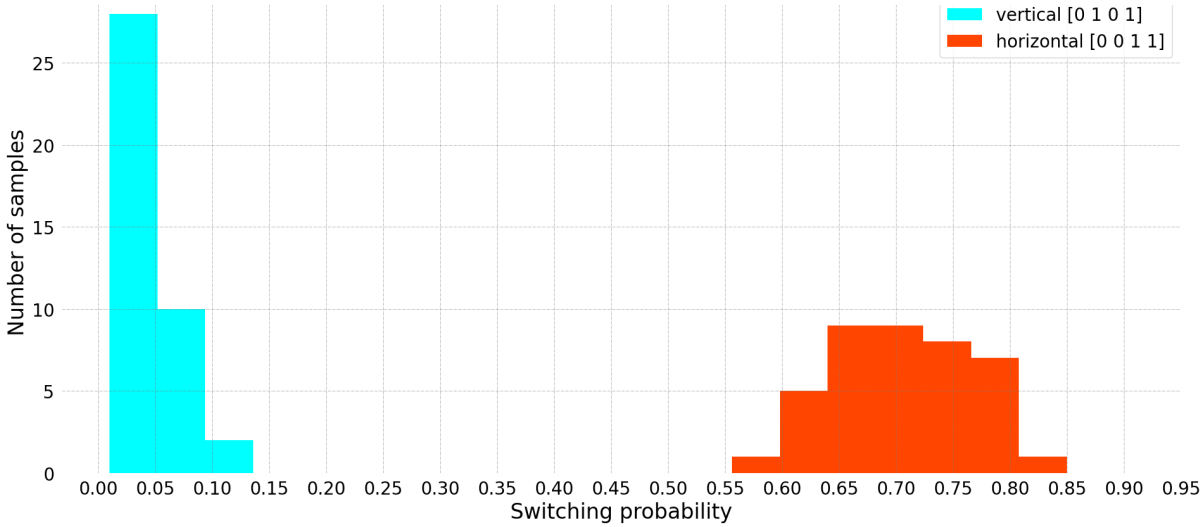



Figure 3.1: Training results of the first experiment with $\mathcal{N} = 100$.

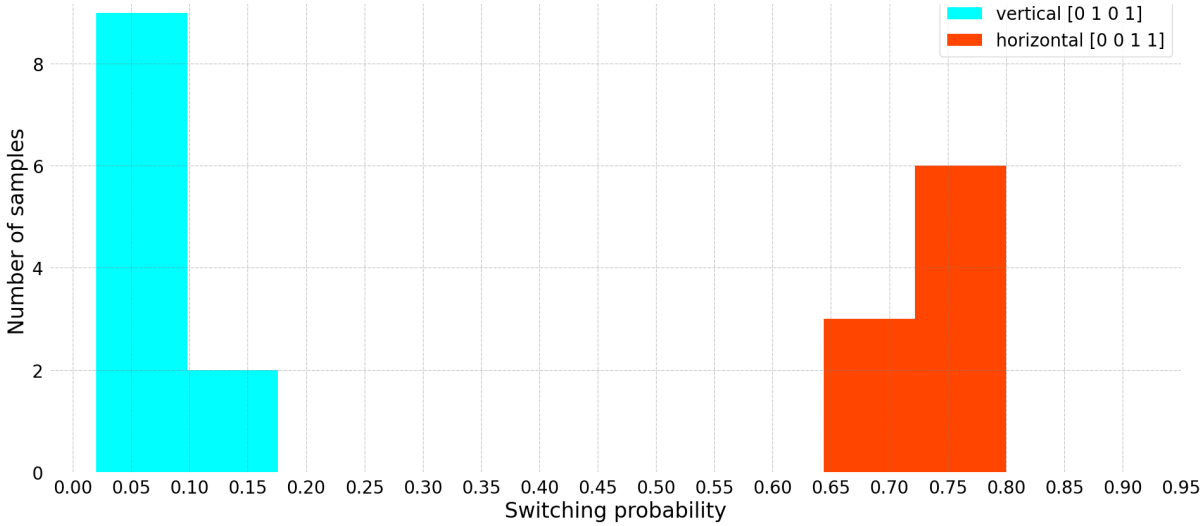



Figure 3.2: Testing results of the first experiment with $\mathcal{N} = 100$.

**In the following, all the switching probabilities were determined with $\mathcal{N} = 10$ attempts for time saving reasons.**

#### 3.1.1.b Grid search

The next data set consisted of 100 samples with a 20 % test ratio of 3 and 8 digits of the scikit database. Finding the correct hyperparameters for these more complex sequences was done by the Optuna grid search algorithm. The search went over 250 different combinations of values (each called a trial) from the following grid :

- a magnetic field $B_x$ between 0.5 A and 2.5 A corresponding to a physical range of [8.35, 41.75] mT (1 A is 16.7 mT)
- a maximum voltage $V_{max}$ between 0.3 V and 0.9 V corresponding to a physical range

of [0.15, 0.45] V (physical voltage value divided by 2 compared to the plot 3.3 for setup reasons)

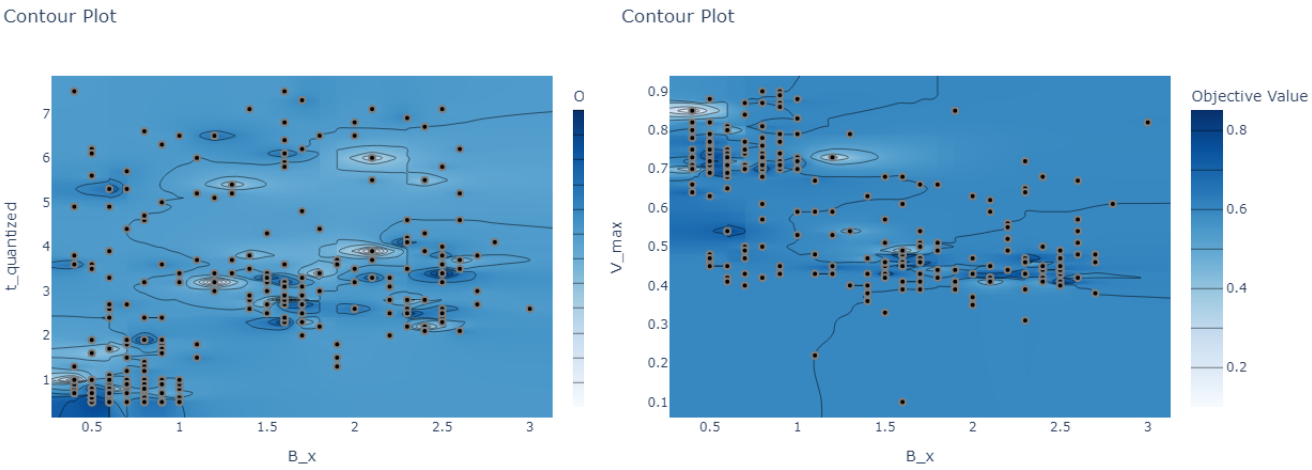
- a quantification step $t_q$ in the range of [0.5, 7.5] ns
- a choice of preprocessing filter between all the available ones (see part 2.2.2)

The best accuracy the algorithm found was 85 % . However, 7 other trials gave 80 % accuracy. On a training dataset of 20 samples, this corresponds to a single sample difference.


Figure 3.3: Accuracy (objective value) as a function of the hyperparameters.

The Optuna search algorithm showed no tangible difference between each pre-processing filter, except for the "short" one which is *a priori* less effective. The accuracy ceiling might also be due to a poor choice of filters.
However, the search showed that longer than 5 ns quantification steps and lower than 20 mV voltages seem to render a lower accuracy. For low voltages, the FL does not switch and therefore the MTJ can't discriminate two distinct signals. For long quantification steps, it doesn't take advantage of heating in the short time scales and might make the FL switch too frequently such that the MTJ still can't separate two distinct signals.
Concerning the influence of the magnetic field, figure 3.4 displaying the contour plots of the hyperparameters shows that we could infer the existence of two regions yielding higher accuracies. In one region, the quantification step can be very short: in that case the weak magnetic field doesn't assist the switching considerably and the switching mechanism is mainly due to heating, different signals will heat the FL differently and it will separate more efficiently different types of signals. In the other region, the quantification step is longer: in that case a stronger magnetic field assists the switching which isn't discriminated through heat anymore.
Simultaneously, a shorter quantification step calls for a higher voltage, so high voltages go with weak fields, while a stronger field assists the switching enough such that low voltages

go with long quantification steps.




(a) (b)

Figure 3.4: Darker blue regions correspond to higher accuracy. Dots are the actual combinations (trials) that were tested. (a) is the quantification step plotted with the field and (b) is the voltage plotted with the field.

For this dataset, the linear regression alone scores 97% accuracy, meaning that the reservoir makes the neural network worse. For binary classification, it seems that the reservoir needs many adjustments before yielding reasonably good results.

### 3.1.2 Multinomial classification

The grid search shows that a scalar switching probability might not be enough to classify the elements of the handwritten digits database. To add features to the training data set, one can assign a switching probability vector to each digit. The regression model can then predict the targets of the testing data set by using not one but several distinct switching probabilities. The following section describes attempts to classify the ten different digits of the scikit database.

#### 3.1.2.a Five emulated MTJs

The first way to test this hypothesis is to emulate MTJs with different hyperparameters each. This effectively expands the size of our reservoir. For this experiment, the data set consisted of 100 samples and 20 % test ratio all filtered by the "averaging" preprocessing filter over sequences of 8 pixels. The data set was sent through the single MTJ 5 times, changing the system's hyperparameters ($t_q$, $V_{max}$, $B_x$) for each time. The resultant accuracy was 10 %, which is exactly random guessing. Moreover, the linear regression alone scores 51 % accuracy. So, the reservoir made it five times worse. The reservoir might not be sufficiently big for the task. The input layer might also lack efficiency : without the input layer, the regression alone scores 90 %. The explanation is simple : the regression gets 64 features to recognize the target vector while the reservoir only gives out 5 switching probabilities to recognize the target vector. However, getting 64 switching probabilities from the reservoir would take too long to produce.

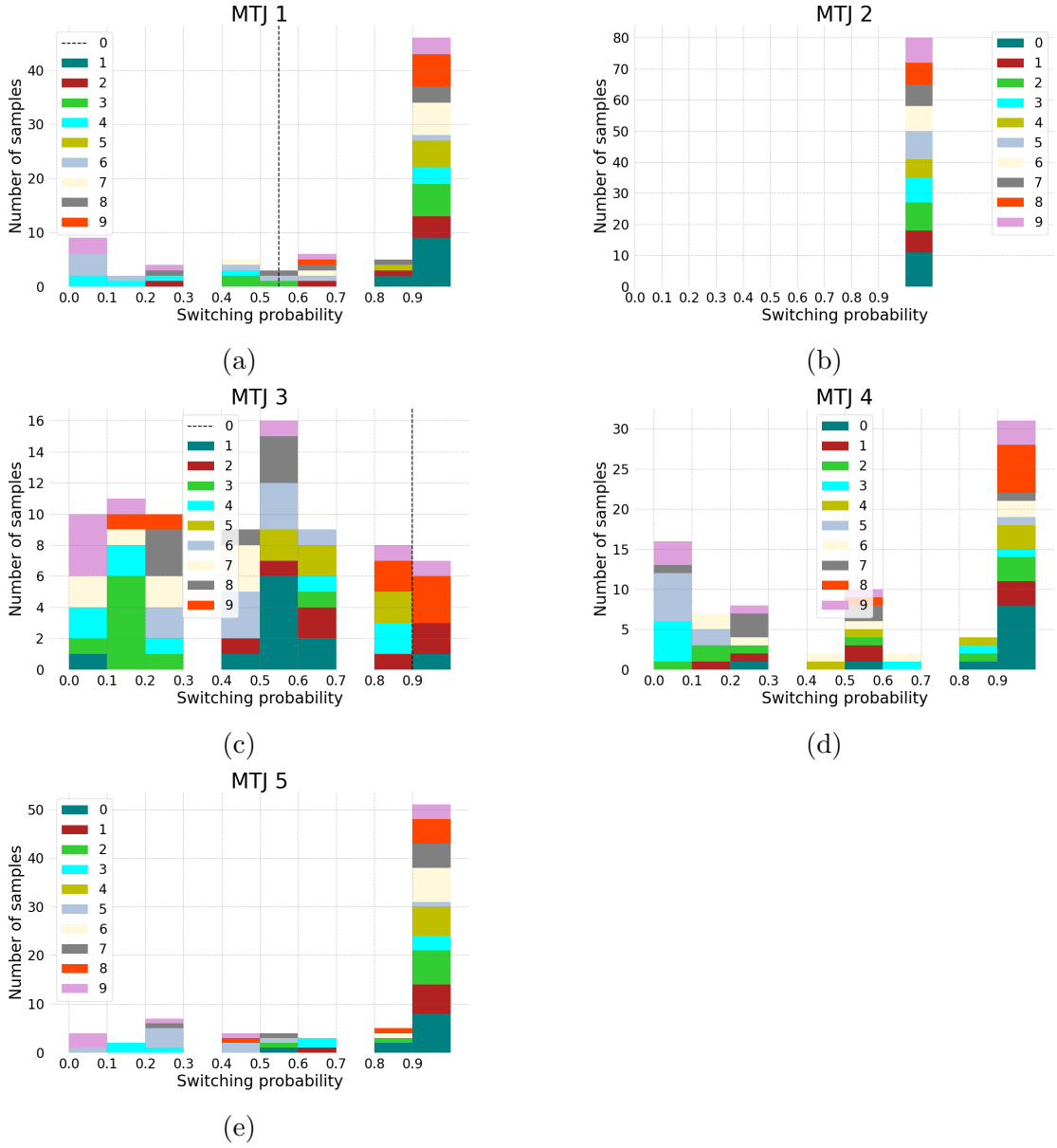



Figure 3.5: 10-class classification : accuracy is 10% (random guessing). Binary classification : if the same "8" arrays yielding $P > 0.55$ in (a) also yield $P > 0.9$ in (c) than we can separate the digits 8 and 3.

#### 3.1.2.b Decomposition of an image in 8 features

Another way of adding features is to use one MTJ but decompose the pixel sequence into 8. For this experiment, the dataset consists of 250 samples with a test ratio of 20 %. Each 1×64 matrix is decomposed into eight 1×8 matrices which all yield a switching probability. This assigns 8 scalars to each digit and the reservoir state is now an 1×8 vector of switching probabilities.
The values chosen for this experiment were those that yielded 80 % accuracy in the Optuna grid search experiment in the weak $B_z$ region.

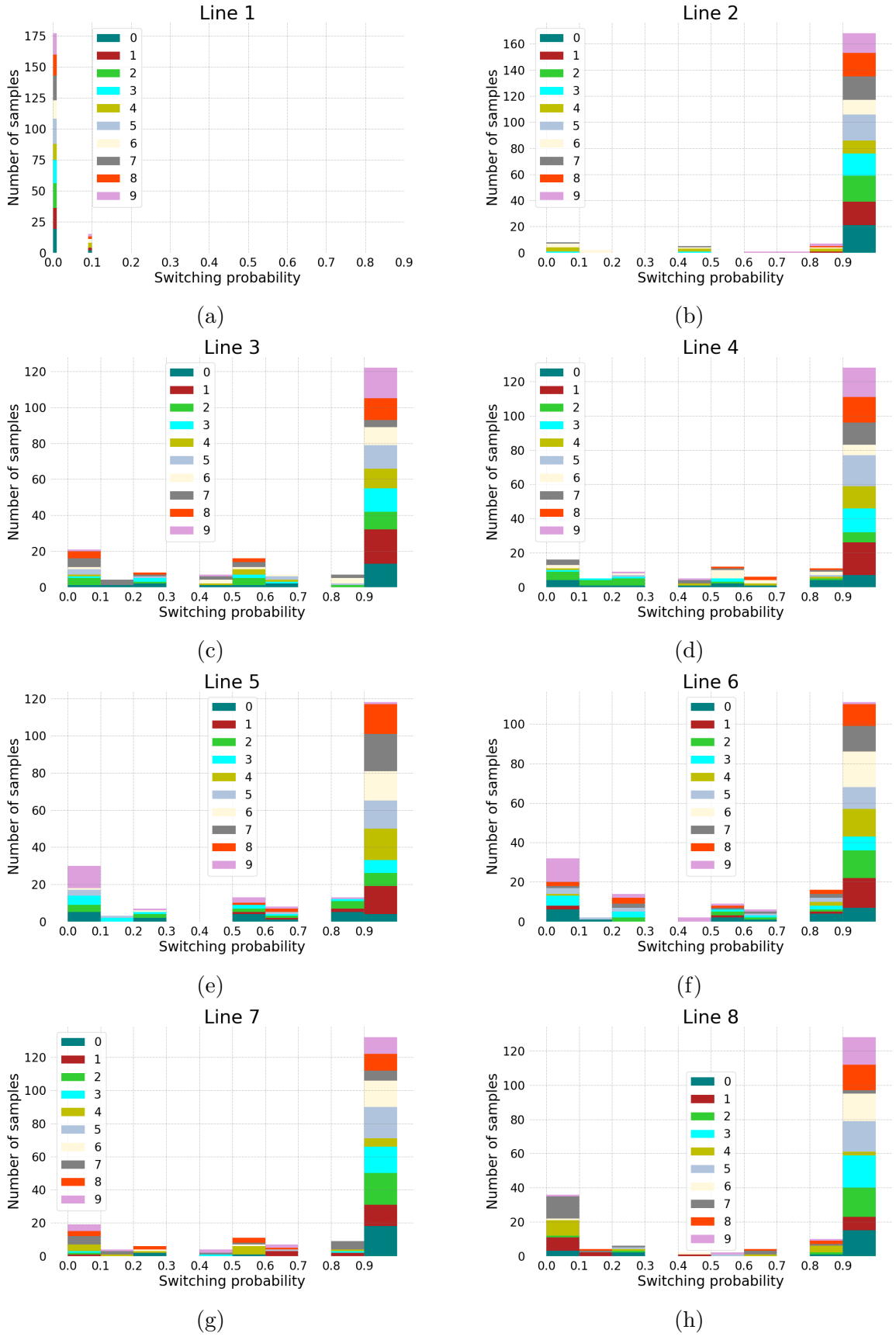



Figure 3.6: Switching probabilities of all 8 lines of all 10 digits in the last experiment. The first line almost never switches because of only black pixels.

The result is an accuracy of 40 % which is much better than the 10 % random guessing of our first attempt to classify all the digits. In figure 3.6, the probabilities are much more spread out than in figure 3.5 (emulated MTJs). This may explain in part why the accuracy is better. The fact that there are 8 features and not 5 also helps increase the accuracy.
Without any reservoir but with the same input layer, the regression scores 56 % accuracy. The reservoir still made it worse but is at least closer to the no-reservoir accuracy.
To improve the accuracy, a grid search would help adjust the MTJ hyperparameters. There is room for improvement in that regard because many of the 8-sequence lines still switch all the time ($\mathcal{P}_{sw} > 0.9$). By expanding the training data set, the accuracy might also increase. With a MNIST database, one could add more features to the dataset since the images are much bigger. This could also increase the accuracy.

## 3.2 Classification with 500 nm wide junction

### 3.2.1 Comparison for binary classification

The results of this subsection are less meaningful than the previous ones, as an error in the code at that time led to a mismatch in some of the classified digits. Nevertheless, the plots still reveal interesting features, which are discussed below.
The 500 nm wide junction switching dynamics should be less dominated by thermal effects than the 80 nm wide junction. Moreover, the nucleation process (see first report about nucleation or [15] and [12]) is longer for a wider junction than for a thinner one. When replacing the 80 nm MTJ with the 500 nm MTJ in the classification task of Scikit handwritten digits 3 and 8 (see 3.1.1.b), the Optuna plots change as follows (see figure 3.7).

The heat time scale dominated region disappears for the 500 nm junction and only the high $t_q$ high $V_{max}$ yields high accuracies. Moreover, the 500 nm junction is more resilient to high voltages than the 80 nm one and was also harder to switch with $x = 0.5$. Adding $x$ to the hyperparameter search space showed that the 500 nm junction needs STT to switch properly and classify the digits. Indeed, the $x$ value yielding the highest accuracies is at 60% ratio.
Unsurprisingly, the maximum accuracy is still below the no-reservoir score of 97%.

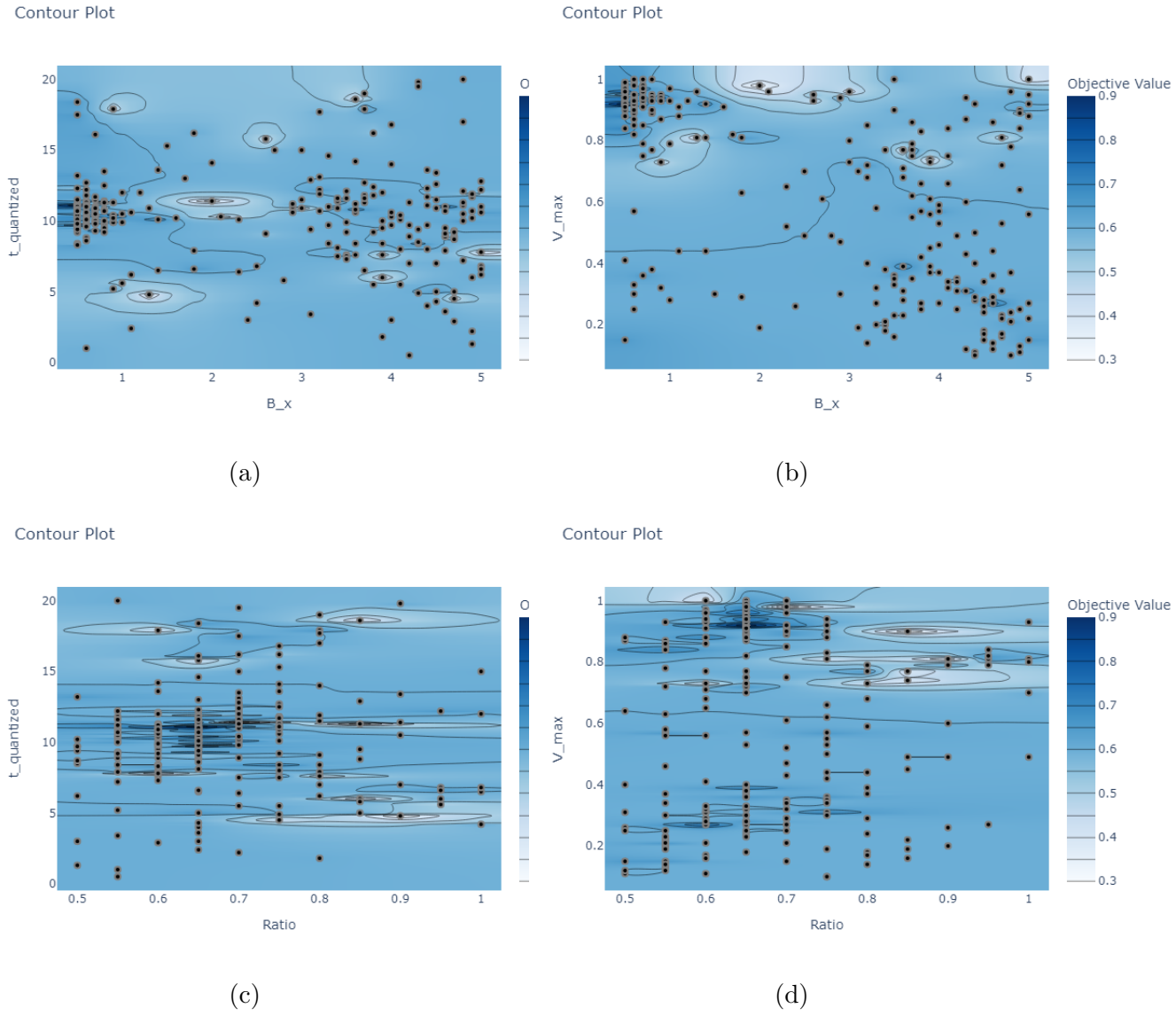



Figure 3.7: Optuna search of the best hyperparameters needed to classify the Scikit handwritten digits 3 and 8 with the 500nm wide junction. Here the $x$ ratio (or 'forward ratio') is an additional degree of freedom in the hyperparameter space.

### 3.2.2 MNIST classification

As mentioned earlier, the output layer scores lower accuracies for multinomial classification. Indeed, for a dataset of 600 MNIST handwritten digits (all 10 digits from 0 to 9) with a test ratio of 20%, the output layer alone scores 33% of accuracy. Meanwhile, the 500 nm wide junction MTJ scores 60% accuracy with the same dataset. This score was obtained with the following hyperparameters, found with Optuna : $t_q$ = 21 ns, $V_{max}$ = 0.42 V, $B_x$ = -2.4 A, $x$ = 0.9 .
These results show once again that the 500 nm junction has more distinctive switching dynamics around the 'high $t_q$, high $V_{max}$' region identified earlier. The switching is dom-

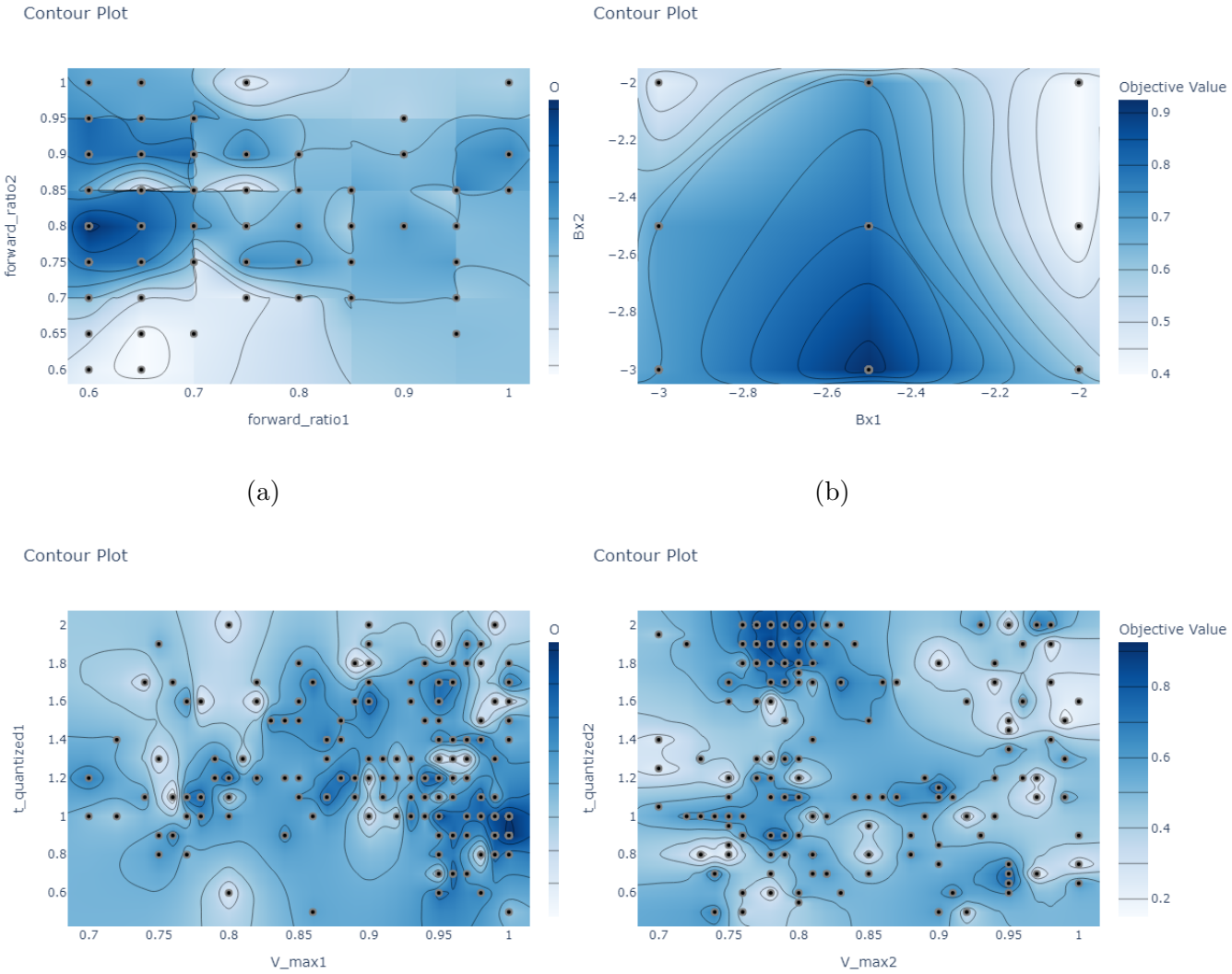
inated by nucleation time, so it is in this region that it will sensitively differentiate two different voltage pulses.
However, 60 % accuracy is still very low compared to what can be done with physical reservoir computing (93% with polaritonics in 2020 on the MNIST database [16]).

### 3.2.3 Period classification

Before trying prediction tasks with the MTJ, an interesting experiment is to classify periodic time series by their periods. The time series to classify were 12-bit long streams of the three following periods : [01], [011] and [0011]. The bit streams are sent as a whole sequence directly to the MTJ and a switching probability is retrieved for each bit stream. The training was done with 16 examples (5 of each) and 4 testing samples (20 total samples with a test ratio of 20 %). The output layer alone scores 100% accuracy on this test. Indeed, they are all linearly separable.

After many trials with one scalar $\mathcal{P}_{sw}$ for each bit stream, the reservoir could not reach an accuracy over 70% because it would either classify a 0011 as a 01 or as a 011 depending on the parameters. Indeed, the idea was to find parameters such that the 01 list never switches, the 011 list always switches and the 0011 list switches only half of the time. The same problem arose when trying to offset the bits by some amount that could reverse the switching. The sensitivity of the reservoir does not seem sufficient to differentiate those 3 different list, even after countless hours of grid search.

The new idea was to try this with two different MTJ settings : each list would get two $\mathcal{P}_{sw}$ : one would classify the 0011 as a 01 and the other would classify it as a 011. The $\mathcal{P}_{sw}$ vector would then be paired to the correct target with the output layer. However, the best result for this method still reached a ceiling at 93% accuracy. However, the plots of this grid search are still interesting 3.8. The optimal combination (darker blue regions) is obtained by using two complementary MTJ settings. In the first case, a relatively long $t_q$ together with a low $V_{\max}$ makes the 00 bits last long enough to cool down the MTJ and erase the memory of the preceding 11 sequence. As a result, none of the subsequent 11 patterns at low $V_{\max}$ are able to trigger switching, and the sequence 0011 is classified as a 01 period. In contrast, with a short $t_q$ and a high $V_{\max}$, the 00 density is insufficient to cool the MTJ, so the memory of the previous 11 is preserved and the high $V_{\max}$ triggers switching. In this regime, the sequence 0011 behaves more like 011, as the MTJ still switches in the same way as for an 011 period. This combination should be enough to classify all three time series. However, the probabilistic nature of the switching prohibits such a deterministic conclusion. The accuracy is hence high but not 100 % as it is without the reservoir.




(a) (b)

(c) (d)

Figure 3.8: Optuna search of the best hyperparameters needed to classify the 12-bit long streams of different periods [01], [011], [0011]. The indices 1 and 2 refer to the first and second switching probability in the $\mathcal{P}_{sw}$ vector coming from the two different MTJ settings. NB : the forward ratio is the $x$ ratio.

After these efforts of classification, it is interesting to explore how the MTJ would perform prediction tasks.

## 3.3 Prediction with 500 nm wide junction

This section displays the results of prediction tasks.

### 3.3.1 Simple periodic predictions

Here, the objective is to predict the $13^{th}$ bit of each of the 12-bit long periodic time series:

$$\mathbf{01}0101010101 \tag{3.1}$$

$$\mathbf{011}011011011 \tag{3.2}$$

$$\mathbf{0011}00110011 \tag{3.3}$$

The output layer alone can predict the $13^{th}$ bit of the 01 stream with an RMSE of $1.1 \times 10^{-16}$ (essentially 0, perfect prediction). However, the RMSE score for the 011 stream is 0.51. Indeed, if the current bit is a 0, the subsequent is a 1, but if it is a 1, the subsequent bit is either a 0 or a 1 with 50% chance. The linear relationship does not help the output layer alone to predict correctly the subsequent bit. The reservoir becomes helpful in this case.

#### 3.3.1.a Prediction sequence by sequence

Trying to send a sequence of bits (cf. 2.3) and training the model on the subsequent bit is counterproductive. A long sequence will more likely switch the MTJ and does not save any training time as one still needs to train the model to predict a bit (or bit sequence) a certain number of times before testing, and this number is incompressible for the moment. The test for the 01 period scores a $1.1 \times 10^{-16}$ RMSE but scores at best a 0.2 RMSE for both the 011 and 0011 periods. The same scores are picked up when trying to add an offset to the time series.

#### 3.3.1.b Prediction bit by bit

The bit-by-bit approach inside the artificial MTJ architecture (cf. 2.9) finds much better results. Indeed, the "memory" of the past bits is conserved inside the channels. By using just two channels : one with $(1, 0)$ coefficients and the other with $(1, 0.45)$ coefficients, the RMSE score went down to $1.1 \times 10^{-16}$ for all three time series, which is essentially a perfect prediction.
To predict more than the $13^{th}$ bit, the inductive method showed the same result: by inserting the predicted bit as a pulse into the architecture again and predicting the $14^{th}$ bit the RMSE still scored $1.1 \times 10^{-16}$. The RMSE score stayed the same for 10 more predicted bits but no further experiment was made to know if the score remained the same further in the prediction. *A priori*, once one period is predicted completely, if the RMSE score stays the same, then it should stay the same for all the subsequent periods. These results are way better than the output layer alone and are the best possible for this time series : a perfect prediction of each period.

### 3.3.2 Lorenz attractor prediction

Following the progression suggested in the benchmarking literature, it is common practice to move from simple periodic signals to more complex periodic dynamics [10], and only thereafter to attempt the prediction of chaotic systems, such as the one-dimensional Mackey–Glass equation [11]. In the present work, however, the only challenging prediction task tested on the proposed reservoir was the Lorenz attractor, a chaotic three-dimensional dynamical system. Reservoir computing has been shown in theory and practice to be capable of predicting such chaotic trajectories [17, 18].

The Lorenz system, originally derived from simplified atmospheric convection equations, is governed by a set of three coupled nonlinear differential equations.

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x), \\ \frac{dy}{dt} &= x(\rho - z) - y, \\ \frac{dz}{dt} &= xy - \beta z, \end{aligned} \tag{3.4}$$

where $\sigma$ is the *Prandtl number*, $\rho$ is the *Rayleigh number*, and $\beta$ is a *geometric factor*. The classical parameter values that generate the canonical chaotic Lorenz attractor (and the ones used here) are :

$$\sigma = 10, \quad \rho = 28, \quad \beta = \tfrac{8}{3}.$$

This system exhibits chaotic dynamics characterized by extreme sensitivity to initial conditions. In our implementation, the $x$, $y$, and $z$ components of the Lorenz system time series were encoded as the three voltage levels of a single input pulse (see 2.3). The system was simulated with a time step of 0.1 seconds. After feeding 20 consecutive data points for training, the model was tasked with predicting the $21^{\text{st}}$ point. The obtained prediction scored a RMSE of 0.02. While this value does not represent a perfect reconstruction of the trajectory, it is a promising result for a first attempt at chaotic time-series prediction using this architecture.
The optimal hyperparameters were determined using Optuna. The resulting values were: $V_{max} = 0.86$ V, $t_q = 6.5$ ns, $x = 0.5$, and $B_x = -2$ A. In addition, the architecture employed eight distinct channel coefficients, given by: $(1, 0.17)$, $(1, 0.26)$, $(1, 0.34)$, $(1, 0.43)$, $(1, 0.51)$, $(1, 0.60)$, $(1, 0.68)$, and $(1, 0.77)$. These multiplicative factors indicate that STT contributed substantially to the switching dynamics.
It should be emphasized, however, that these results stem from a single exploratory trial and must therefore be interpreted with caution. A more systematic investigation, including multiple runs with different random seeds and parameter sensitivity analysis would be required to draw firm conclusions regarding the reservoir's ability to generalize to chaotic dynamics.

# Two-terminal MTJ

This information might not be worth mentioning in this report but is mentioned for completeness sake.
We tried to do the same experiments with only STT on the probe station, but no result was conclusive (last days of internship) and the MTJ switched very seldomly so there were no clear distinction between the pulses.

# Conclusion and outlook

In this report, we explored the implementation of reservoir computing using magnetic tunnel junctions as nonlinear physical reservoirs. By designing encoding strategies, training protocols, and hyperparameter optimization procedures, we tested the capacity of MTJs to perform both classification of handwritten digits and prediction of time series. The results indicate that linear models perform significantly better for simple binary tasks, and that our approach, relying on the switching probability of the MTJ, is not yet well-suited for these problems. Furthermore, this approach does not perform optimally for multinomial classification, although it still achieved higher scores than the linear models in these scenarios. Nevertheless, our MTJ-based reservoirs appear to show considerable potential for time series forecasting, achieving perfect predictions for simple periodic bitstreams and promising results for chaotic signals.

The experiments highlighted the dependence of the reservoir's performance on device hyperparameters such as pulse duration, applied voltage, magnetic field, and forward ratio. Bayesian optimization of the grid search provided useful insight into regions of parameter space where switching dynamics enhance separability of input patterns. Although current accuracies remain below state-of-the-art physical reservoirs, the findings suggest that MTJs can capture meaningful nonlinear dynamics and that the reservoir performance improves when richer feature representations are employed.

Improvements to these experiments could be achieved by increasing the number of MTJs in a purpose-designed architecture, enabling better performance across all tasks. Finally, further improvements could lie in exploring approaches beyond the switching probability.

# Abbreviations

- MTJ : magnetic tunnel junction
- FL : free layer
- P : parallel state of the MTJ
- AP : anti-parallel state of the MTJ
- SOT : spin-orbit torque
- STT : spin-transfer torque
- VCMA : voltage-controlled magnetic anisotropy
- AWG : arbitrary waveform generator
- PXI : peripheral component interconnect extension for instrumentation

# Bibliography


[1] Herbert Jaeger. "The "echo state" approach to analysing and training recurrent neural networks-with an erratum note". In: *Bonn, Germany: German national research center for information technology gmd technical report* 148.34 (2001), p. 13.

[2] Min Yan et al. "Emerging opportunities and challenges for the future of reservoir computing". In: *Nature Communications* 15.1 (2024), p. 2056.

[3] Jaideep Pathak et al. "Using machine learning to replicate chaotic attractors and calculate Lyapunov exponents from data". In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 27.12 (2017).

[4] Giuseppe Alessio D'Inverno and Jonathan Dong. "Comparison of Reservoir Computing topologies using the Recurrent Kernel approach". In: *Neurocomputing* 611 (2025), p. 128679.

[5] Matthew Denton and Herman Schmit. "Direct spatial implementation of sparse matrix multipliers for reservoir computing". In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2022, pp. 1–11.

[6] Claudio Gallicchio. "Sparsity in reservoir computing neural networks". In: *2020 international conference on INnovations in intelligent SysTems and applications (INISTA)*. IEEE. 2020, pp. 1–7.

[7] Nils Schaetti, Michel Salomon, and Raphaël Couturier. "Echo state networks-based reservoir computing for mnist handwritten digits recognition". In: *2016 IEEE Intl conference on computational science and engineering (CSE) and IEEE Intl conference on embedded and ubiquitous computing (EUC) and 15th Intl symposium on distributed computing and applications for business engineering (DCABES)*. IEEE. 2016, pp. 484–491.

[8] Mu-Kun Lee and Masahito Mochizuki. "Handwritten digit recognition by spin waves in a Skyrmion reservoir". In: *Scientific reports* 13.1 (2023), p. 19423.

[9] Daniel Auge et al. "A survey of encoding techniques for signal processing in spiking neural networks". In: *Neural Processing Letters* 53.6 (2021), pp. 4693–4710.

[10] Hugo Cisneros, Tomas Mikolov, and Josef Sivic. "Benchmarking learning efficiency in deep reservoir computing". In: *Conference on Lifelong Learning Agents*. PMLR. 2022, pp. 532–547.

[11] Kilian D Stenning et al. "Neuromorphic overparameterisation and few-shot learning in multilayer physical neural networks". In: *Nature Communications* 15.1 (2024), p. 7377.

[12] Eva Grimaldi et al. “Single-shot dynamics of spin–orbit torque and spin transfer torque switching in three-terminal magnetic tunnel junctions”. In: *Nature nanotechnology* 15.2 (2020), pp. 111–117.

[13] James Bergstra et al. “Algorithms for hyper-parameter optimization”. In: *Advances in neural information processing systems* 24 (2011).

[14] Shuhei Watanabe. “Tree-structured parzen estimator: Understanding its algorithm components and their roles for better empirical performance”. In: *arXiv preprint arXiv:2304.11127* (2023).

[15] Viola Krizakova et al. “Spin-orbit torque switching of magnetic tunnel junctions for memory applications”. In: *Journal of Magnetism and Magnetic Materials* 562 (2022), p. 169692.

[16] Dario Ballarini et al. “Polaritonic neuromorphic computing outperforms linear classifiers”. In: *Nano Letters* 20.5 (2020), pp. 3506–3512.

[17] Lauren A Hurley and Sean E Shaheen. “Reservoir computing with large valid prediction time for the Lorenz system”. In: *arXiv preprint arXiv:2508.06730* (2025).

[18] Nicholas W Landry et al. “A theoretical framework for reservoir computing on networks of organic electrochemical transistors”. In: *arXiv preprint arXiv:2408.09223* (2024).