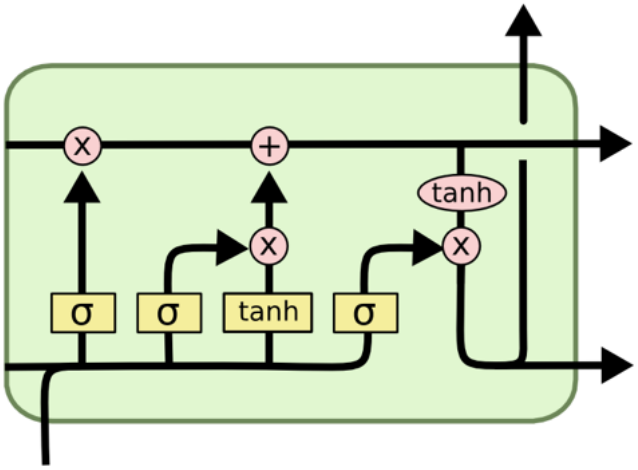
École Polytechnique Fédérale de Lausanne
School of Engineering

Exploring the use of digital feedback and neural networks for tackling time domain problems

Ambrine Douhane
Microengineering Master Student
Spring 2023



LSTM cell [1]

Prof. Demetri Psaltis — Optics Laboratory
Prof. Christophe Moser — Laboratory of Applied Photonics Devices
PhD Student Ilker Oğuz — Supervision

Abstract

This report presents the application of LSTM and GRU neural networks on raw and processed data obtained from an optical experiment for video classification. The objective of the study was to find the most effective approach for dealing with memory in video classification tasks using a data-set consisting of up to 15 frames per video.

The results revealed that both LSTM and GRU networks did not perform well in classifying the videos. They either failed to achieve meaningful classification or showed significant over-fitting, indicating that the data-set was unsuitable for these methods. As an alternative, an exploration of utilizing the feedback output of the previous image was conducted to improve memory retention and classification accuracy.

The findings demonstrate that the approach involving SOLO outputs outperformed LSTM and GRU networks in video classification. The study further investigated the parameters of the feedback mechanism to optimize accuracy. These results provide valuable insights into memory-enhancing techniques for video classification tasks.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context

Transforming data-sets by optical interactions have shown to be improving the classification performances with neural networks. When dealing with multimode fibers (MMFs), which support a large number of propagating modes, the resulting output is influenced by both linear and nonlinear effects. This is due to the interference among different modes and their interactions with the surrounding medium.

In this research project, emphasis is placed on an optical setup characterized by its time-invariant nature. The absence of time dependence in all the optical elements used does not pose any issues in scenarios such as image classification. However, the incorporation of memory is required for certain recognition tasks. For instance, the determination of an object's motion necessitates prior knowledge of its position, which changes over time.

To address this need for memory, two approaches are being explored:

a) The computational method: in this approach, the previous data is stored and processed digitally using software-based solutions. The storage and retrieval of past information enable enhanced object recognition.

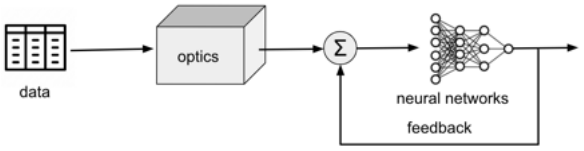


Figure 1: Computational method

b) The optical method: here, the investigation is focused on optical means of storing previous data. Specifically, the utilization of spatial light modulators is being explored. These SLM are designed to orient their dipoles relative to the direction of the incident field. By incorporating such optical elements, the goal is to store and retrieve relevant data optically, thereby facilitating improved object recognition.

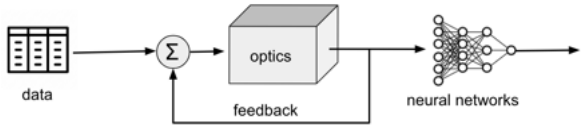


Figure 2: Optical method

By employing both computational and optical techniques, the recognition capabilities of the optical system can be enhanced, enabling the tackling of complex tasks that require memory. The research aims to contribute to the advancement of machine learning-based optical recognition systems and their practical applications.

1.2 Basic knowledge on machine learning

Neural networks, inspired by the human brain, consist of interconnected layers of artificial neurons. To enable accurate image and video classification, these networks are trained on labeled

data-sets. In the training phase, the neural network learns patterns and features within the data through a process known as training. This involves adjusting the network's parameters based on the provided labeled examples, enabling it to make predictions on unseen data.

Once the model is trained using the training set, it is essential to evaluate its performance on a separate set of data known as the validation set. The validation set consists of examples that the model has not been exposed to during the training phase. This evaluation process helps assess the model's ability to generalize and accurately classify new images and videos.
Following the training and validation stages, the trained model can be utilized for classifying unseen data. This involves inputting new images or videos into the trained model, which generates predictions or probabilities for each class. These predictions reflect the model's confidence in its classification decisions, enabling accurate and reliable classification of new and previously unseen data.

Keras [6] is a user-friendly open-source library for building and training deep learning models. With Keras, one can define the architecture of the network, specify the training parameters, and feed the labeled data to the model. This library handles the optimization process, adjusting the network's weights and biases to minimize the prediction errors during training. It has been used in the following work.

1.3 Initial data set

One of the challenges in this work is to organize the data-set before providing it as input to the model. Therefore, it is crucial to have a comprehensive understanding of its specificities. The data can be in various formats such as .txt, .csv, .xlsx, or json. The initial step of the project involves extracting and preparing the data to facilitate further analysis and utilization on the Google Colab platform. This preparatory stage ensures that the data is properly structured and ready for subsequent processing and modeling tasks.

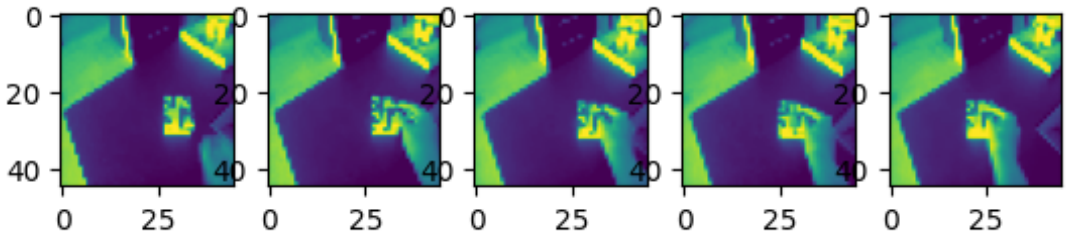
```
from google.colab import drive
drive.mount('/content/gdrive')

!cp gdrive/My\ Drive/data/sthsthPush45_4cl.npz .

prevPack ='sthsthPush45_4cl.npz'
f = np.load(prevPack)
labels=f['labels']
data=f['data']
```

Figure 3: Python code for accessing the data and label

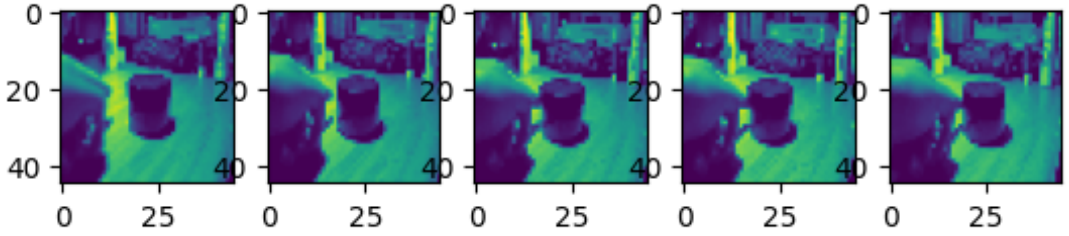
The data-set consists of two main components: the 'data' array and the 'labels' array.

The 'data' array is structured with a shape of (2, 2000, 15, 45, 45), representing a multidimensional collection of samples. Each one of the $2 \cdot 2000$ samples contains 15 frames, where each frame comprises a single channel because the images are grey scale. Then, each image is made of 45x45 pixels.

On the other hand, the 'labels' array complements the 'data' array by providing categorical labels for the corresponding samples. These labels serve as class identifiers or categories associated with each sample, enabling the classification and analysis of the data. The two categories in this classification are: items moving from left to right and items moving from right to left (see Fig. 4).



(a) Item moving from right to left



(b) Item moving from left to right

Figure 4: Five frames of two random samples from our data set

2 Work on LSTM

2.1 Theoretical background

An LSTM (Long Short-Term Memory) network is a type of recurrent neural network (RNN) that addresses the vanishing gradient problem and is particularly effective in modeling and processing sequential data. It consists of memory cells and gating mechanisms that regulate the flow of information through the network.

The basic building block of an LSTM is the memory cell, which is responsible for capturing and storing information over time. The cell has a cell state (also known as the memory state) that acts as a long-term memory and retains information from previous time steps. The cell state can be selectively updated, allowing the network to remember or forget specific information.

LSTMs incorporate three types of gates that control the flow of information: the input gate, forget gate, and output gate. These gates are neural network layers that determine how much information should be let through at each time step.

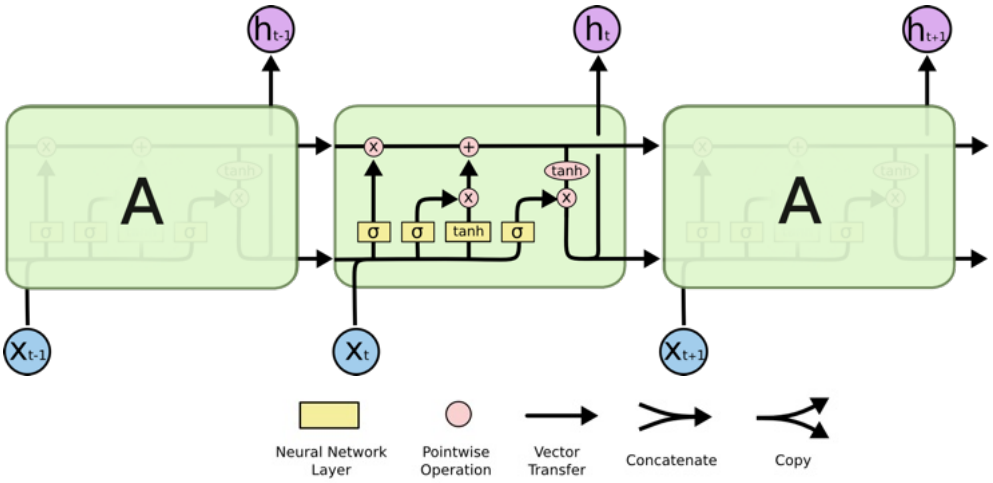


Figure 5: LSTM network based on three interacting cells

The input gate controls the amount of new information to be added to the memory cell. It decides which parts of the input should be stored in the cell state. The forget gate determines which information from the previous cell state should be discarded. It regulates the extent to which the previous cell state affects the current state. The output gate controls the information to be outputted from the cell state to the next layer or the final prediction.

The gates in an LSTM network are composed of sigmoid activation functions that produce values between 0 and 1. These values represent the proportions of information to be passed through the gates. The sigmoid function allows the gates to learn to open or close based on

the relevance of the information.

By utilizing these memory cells and gating mechanisms, LSTMs can capture long-term dependencies in sequential data. They can learn to retain important information over many time steps and selectively forget irrelevant details. This ability to retain and propagate information over time makes LSTMs particularly suitable for tasks involving sequential data, such as speech recognition, language translation, and sentiment analysis.

2.2 CNN with LSTM

2.2.1 Plan

A first try has been performed for adapting the code of a standard model of Fig. 22 to our data described in Section 1. Cette adaptation a consisté en adapter le model à la forme des data, à la nature des labels,

2.2.2 Encountered issues and their resolution

Initially, the accuracy was found to be around 0.5. Since the number of classes is 2, there was no learning happening, indicating a faulty procedure. It is possible that either the labels and data were in the wrong format or were not properly synchronized, or the model itself was not suitable for our data. The following paragraphs present the tests and attempts made to identify the issues that hindered the experiment.

Data and labels:
A first data inspection was conducted to ensure the data's usability. A cursory examination confirmed that the data was suitable for further analysis. Notably, the images comprising the videos exhibited no anomalies or irregularities. Randomly displaying the videos from the X_{train} and X_{test} lists confirmed that they align with the expected content, further validating the integrity of the data-set.

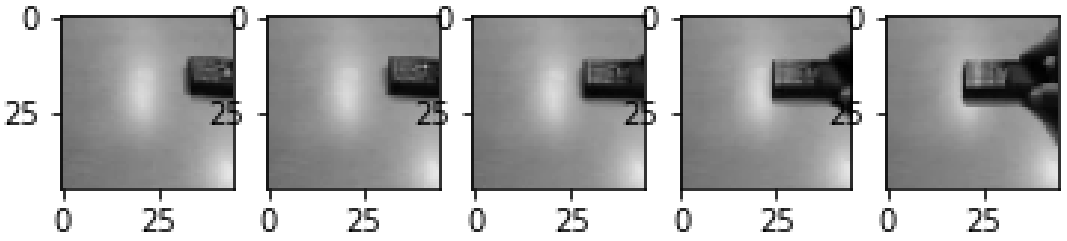


Figure 6: Are the data frames in the good shape ? - plot of 5 frames of a random video of X_{test}

The first test aims to display the distribution of labels. The expected outcome is that all labels should be equal to 0 for the first 2000 samples, and equal to 1 for the subsequent 2000 samples. Upon conducting the test, it was observed that the label distribution aligns precisely with the

expected pattern (see Fig. 7), confirming the correct labeling of the data-set.

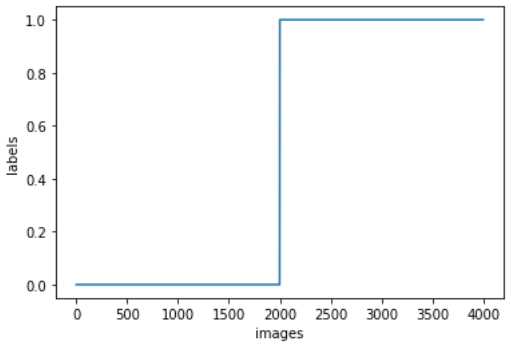


Figure 7: Are the labels well spread ? - Labels plot

In the case of a classification problem with two integer labels representing distinct classes, it is necessary to perform one-hot encoding. This involves transforming the labels from a numpy array of size $(4000, 1)$ to an array of dimensions $(4000, num\_classes)$, where each label is represented by a vector with a one in the corresponding class index and zeros in all other indices. This encoding scheme ensures that the model can properly interpret and classify the data based on the distinct classes. This process can be achieved quite straightforwardly, as demonstrated in Fig. 25 in the Appendix.

Model correction and simplification:
To improve the accuracy of our model, it was deemed necessary to simplify it. With this objective in mind, adjustments were made to the model architecture. Firstly, the decision was made to reduce the number of images per video from 15 to 3, evenly spaced at temporal intervals. This reduction was intended to streamline the input data and eliminate potential noise or redundancy. Additionally, the number of LSTM units in the model was decreased (see Tab. 1), and the number of training epochs was increased from 2 to 30.

LSTM number of units	Accuracy
256	0.541
128	0.540
64	0.49
32	0.541

Table 1: Number of LSTM units in the model and their according performances

Despite these modifications, the impact on the results was not substantial, and the accuracy remained around 0.5. It became evident that further simplifications were warranted. However, a deliberate choice was made not to diminish the resolution of the images. The resolution was already relatively low, and further reduction had the potential to compromise the model's

performance on this specific data-set.

In pursuit of further model simplification, a modification was made to transition from the VGG16 architecture to a Conv2D model, knowing that Conv2D is a single layer that performs 2D convolution, whereas VGG16 is a complete CNN architecture that utilizes multiple Conv2D layers in its design, and is thus more complex.

2.3 CNN without LSTM

To identify or locate the source of the problem, an effective approach can also involve the subdivision of the model into multiple sub-models that can be tested individually. This led to the removal of the LSTM layers. By doing so, the data also had to be transformed, transitioning from videos to single frames. The middle image was chosen to test the new LSTM-free model. The Python code that enabled this operation can be found in the Appendix in Fig. 26. Subsequently, it was observed that the validation accuracy increased significantly, deviating from 0.5. Thus, it was understood that the issue specifically originated from the LSTM layers present in the previous model.

With the objective of video classification, further attempts at classification were pursued by utilizing another code found on GitHub, written by Jerinkantony. While it is true that the problem could be localized, it is also true that the underlying causes have not been identified, which is frustrating. However, a decision was made to employ an LSTM network, which explains why the study once again focused on a model composed of LSTM units even if the first tries appeared unsuccessful.

2.4 Back to CNN with LSTM

The new code was implemented using the script shown in Fig. 27 in the appendices, and a graphical representation of this model is provided in Fig. 8.
The performance of this model is shown in Fig. 9. It can be observed that the validation accuracy is approaching unity, which is indicative of favorable performance. The model achieves accuracy surpassing 0.5, demonstrating its ability to correctly classify videos into their respective classes.

2.4.1 Overfitting

However, on Fig. 9 there are also distinguishable trends that indicate the presence of overfitting. For example, it is evident that the error on the training data is significantly lower than that on the validation data, suggesting that the model has excessively learned from the training data. Another indicator of overfitting is the disparate growth dynamics between training accuracy and validation accuracy across epochs.

The model is relatively straightforward, consisting of an input layer utilizing VGG16 as the base model for feature extraction, leveraging pre-trained weights from ImageNet. Subsequently, each frame of the input video sequence undergoes independent processing by the VGG16 base model to extract visual features. These encoded frames are then sequentially processed by an LSTM layer, enabling the capture of temporal dependencies within the video sequence. A dense layer with ReLU activation follows, facilitating the learning of intricate patterns and relationships. Ultimately, another dense layer with softmax activation generates predicted class probabilities for the video sequence. To summarize the visual content of the video frames, global average pooling is applied to the output of the CNN base model.

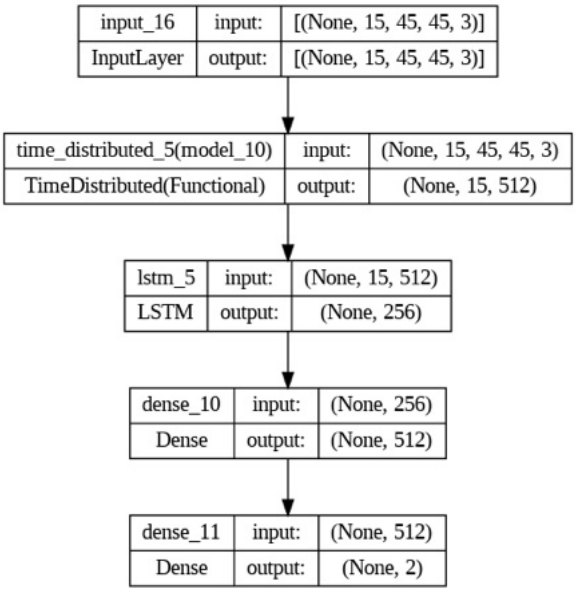


Figure 8: Graphical representation of the new model using LSTM

Overfitting occurs when a model performs well on the training data but struggles with unseen data, indicating an imbalance between memorizing the training data and capturing the underlying patterns and relationships between the input features and their corresponding labels.

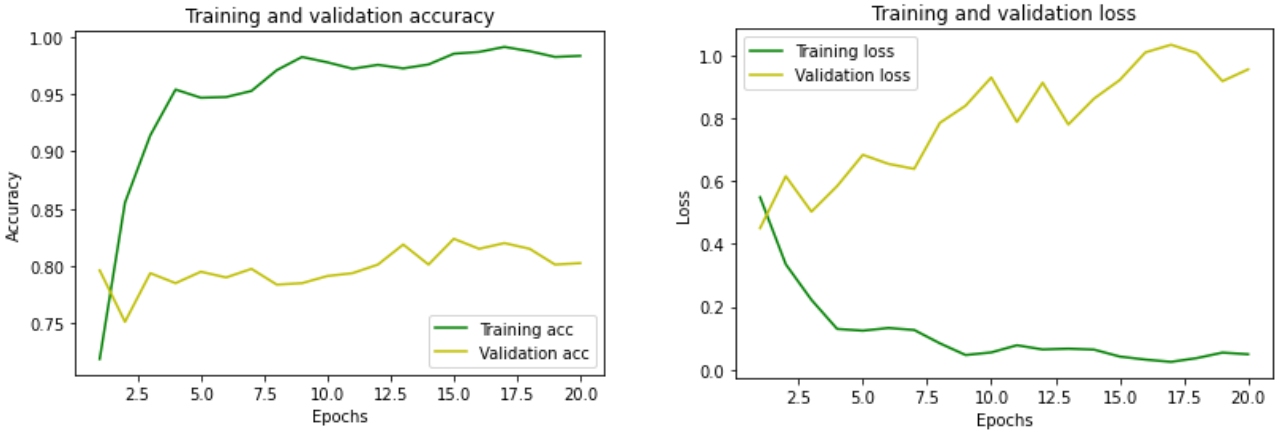


Figure 9: Performance of this new LSTM model

To mitigate overfitting, various strategies have been attempted. Firstly, the model was simplified by reducing the number of LSTM units from 256 to 32 (see Fig. 35) and decreasing the number of images per video from 15 to 5 (see Fig. 36). However, these modifications did not effectively alleviate overfitting, as evidenced by the nearly unchanged performance of the model post-adjustments.

Both the additions of a layer of regularization to the dense layers and a high rate drop-out to prevent overfitting were also attempted, as shown in the code lines depicted in Fig. 10. Once again, the performance did not exhibit significant changes (see Fig. 37 and **??** in the appendix).

```
from keras.regularizers import l2
hidden_layer = Dense(
                        units=512,
                        activation="relu",
                        kernel_regularizer=l2(0.01)  #L2 regularization
                    )(encoded_sequence)

hidden_layer = Dropout(0.5)(hidden_layer)  #dropout
```

Figure 10: Python code for adding regularization to the dense layers and increasing the dropout rate

In this context, it was deemed irrelevant to employ data augmentation techniques, such as applying random transformations such as rotation, scaling, and flipping to the input data. This is because the essence of our categories, and thus our data, would lose their inherent meaning through such transformations.

2.4.2 Craft on model

Once again, the LSTM portion of the code was removed (see code in Fig. 28) to determine if it was the cause of the model's poor performance. Indeed, as depicted in Fig. 38, removing the LSTM layers effectively reduces a significant portion of the overfitting.

It is observed in Fig. 2 that there exists a substantial discrepancy in the number of parameters between the two models, namely the one incorporating LSTM and the one without LSTM. Consequently, we are confronted with a comparison of two models that are inherently incomparable due to their dissimilarities in parameterization.

	model with LSTM	model without LSTM
trainable parameters	14 714 688	13 128 130
non-trainable parameters	87 682	0
total	14 802 370	13 128 130

Table 2: Number of parameters for the different models

The current approach aims to incorporate the successful CNN network into our data and augment it with LSTM. The code for this in shown in Fig. 29. Subsequently, a comparative

analysis will be conducted to evaluate the accuracy of this combined model (see Fig. 11) in comparison to the previous model (without LSTM) and assess the resultant outcomes.

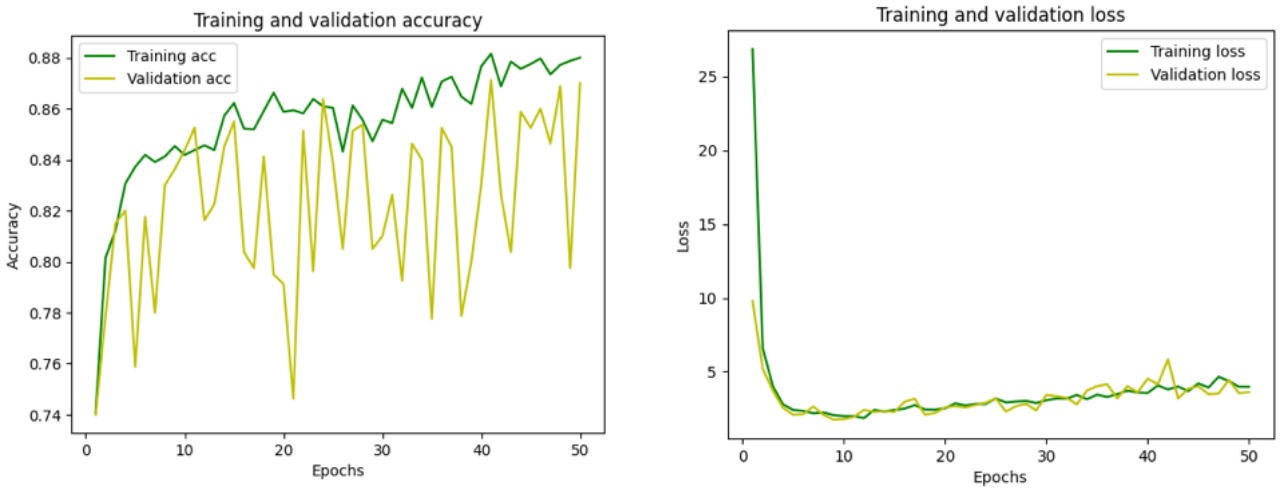


Figure 11: Performance of the new combined model

This has led to unsatisfactory results, notably due to the unexpected fluctuations in validation accuracy. Attempts to mitigate these fluctuations by revising hyperparameters such as learning rate, regularization strength, and number of hidden units have proven unsuccessful.

In fact, the significance of achieving high validation accuracy is not paramount in this context. The crucial message conveyed here is that, with this specific data-set, the integration of LSTM does not contribute to improved results alongside the CNN.

2.5 Results

Several operations have been attempted to utilize LSTM for our video classification. LSTM appears to be ill-suited to our data, maybe due to insufficient training data. Indeed, LSTM models typically require a large amount of training data to effectively capture temporal relationships. Further testing will be conducted using GRU (Gated Recurrent Unit) models.

3 Work on GRU

3.1 Theoretical background

GRU (Gated Recurrent Unit) is a type of recurrent neural network (RNN) architecture that addresses the vanishing gradient problem by capturing long-term dependencies in sequential data. It is similar to LSTM but has a simpler structure with fewer gates and memory cells (see Fig. 43). GRU combines the forget and input gates into a single update gate, reducing computational complexity. Additionally, it introduces a reset gate that allows selective updating of the memory based on the current input and previous hidden state. Both GRU and LSTM excel at capturing long-term dependencies, but GRU is generally computationally more efficient and easier to train due to its simplified architecture.

3.2 New tries

This research focuses on working with processed data, which requires transformation prior to constructing the model. Processed data refers to the data derived from an optical experiment based on a given input of raw data.

3.2.1 Data transformation

Currently, the data is stored in a format denoted as data shape = (12000, 180, 180). In order to utilize the associated labels and facilitate future applications, it is necessary to perform the following transformations. Firstly, the data shape will be modified from (12000, 180, 180) to (12000, 45, 45), as this will prove advantageous for subsequent operations and enhance processing speed. Secondly, the modified data will be further reshaped from (12000, 45, 45) to (4000, 3, 45, 45), where 3 represents the number of frames per video and 4000 represents the total number of videos. Finally, the data and labels, now possessing compatible shapes, can be linked together for further analysis.

3.2.2 Data inspection

The occurrence of zeros in the array poses a potential risk to the experiment, as it signifies a lack of data. Hence, it was imperative to verify the frequency of zeros (as depicted in Fig. 12) prior to conducting any further operations. This yields a proportion of 1.83% of zeros, which is within an acceptable range. The distribution of all data values has been provided in Fig. 44 and appears to be well-distributed.

3.3 Results

The verified data was fed into the model, whose code is provided in Fig. 30, and the classification accuracy exhibited signs of overfitting. To address this, the VGG16 layers were replaced with a pair of trainable convolutional layers (as shown in the code snippet in Fig. 31). The

```
nb_zeros = np.count_nonzero(data3 == 0)
nb_cases = np.size(data3)
ratio=nb_zeros*100/nb_cases
```

Figure 12: Python code for getting the proportion of zeros in the data

use of a pre-trained VGG as the convolutional preprocessing step could be the source of the problem, considering that it is trained on natural images from the Imagenet data-set, while our data-set belongs to a distinct domain. As depicted in Fig. 13, the results demonstrate a flat validation accuracy of 0.5 across epochs. The performance remains unsatisfactory.

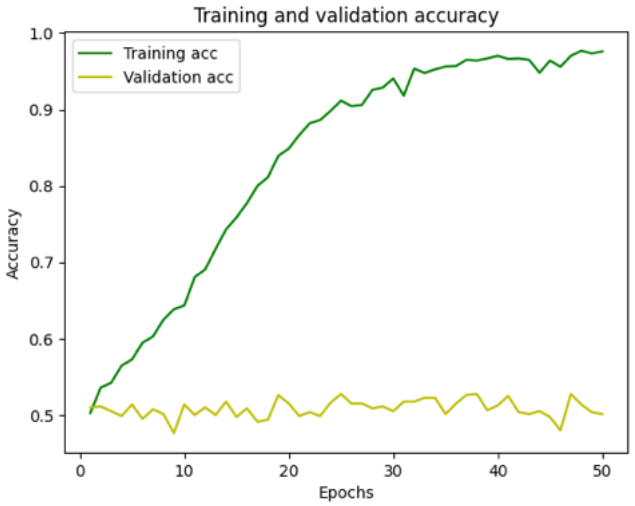


Figure 13: Performances obtained with the model based on GRU

Ultimately, even a GRU-based model, which is known to be simpler than an LSTM-based one, does not yield favorable results. For this reason, another approach is being attempted to address the memory aspect in video data.

4 Another approach for dealing with memory

In the continuation of this work, the data is modified so that when it is provided as input to a model, it not only contains information about the input videos in our data-set but also includes information related to memory.

4.1 Different ways of dealing with the memory

Two approaches to utilizing memory have been tested. For the sake of clarity, we denote S as the input data to the model, which comprises the data used for training and testing. On the other hand, D represents the raw data that solely contains information about the videos. Additionally, O denotes the output of the presumed optical system in response to a stimulation. Initially, the approach involved providing the model with an input consisting of an image for each video, which encompasses the information from the current video frame as well as the two preceding frames:

$$S(x, y, t) = D(x, y, t) + a \cdot D(x, y, t-1) + b \cdot D(x, y, t-2) \tag{1}$$

where a and b are real values, and x and y represent the columns and rows of the images, t is the image time in the video.

Another idea was to provide the model with an input image that contains the information of the current video frame, along with the output image of the previous frame from the same video:

$$S(x, y, t) = D(x, y, t) + a \cdot O(x, y, t-1) \tag{2}$$

4.2 Optical experiment

4.2.1 Proper optical experiment

The analysis of the output from this experiment can be used to enhance the performance of a video classification model.

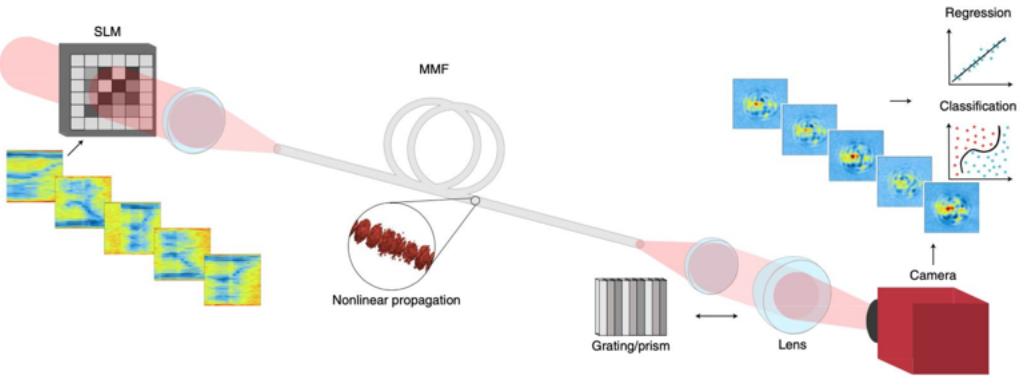


Figure 14: Illustration of the neuromorphic computing architectures and the experimental set-up [2]

This experiment utilizes a multimode fiber and enables the propagation of laser light through a series of optical instruments. The setup of OL's Experiment 1 from can be found in Fig. 47. More specifically, in the image of Fig. 14, the experimental configuration can be observed, which includes a Spatial Light Modulator (SLM) used to encode information onto laser pulses, a Multimode Fiber (MMF) employed for nonlinear propagation, and an imaging system for information decoding.

At the time when the experiment was intended to be performed, the laser used in this experiment malfunctioned. Consequently, an attempt was made to digitally replicate these two experiments.

4.2.2 First approximation

A random matrix M was used for modelling the non-linear transformation in the experiment. Thus, the output data O is given by:

$$O(x, y, t) = \sigma(M \times D(x, y, t)) \tag{3}$$

where σ is a sigmoid function, that compresses the output to a range of 0 to 1, enabling probability interpretation in classification tasks.
More precisely, since for the sigmoid, the values are respectively very close to 0 and 1 for $x < -5$ and $x > 5$, we introduce an scaling factor α and play on it to kind of normalize the range of x.

$$\sigma(\alpha, x) = \frac{1}{1 + e^{-\alpha x}} \tag{4}$$

Setting α:
8,100,000 values are stored in the np.array that contains $\alpha \cdot M \times S(x, y, t)$ values.

α	Number of > 5 values in $\alpha \cdot M \times S(x, y, t)$	Number of < -5 values in $\alpha \cdot M \times S(x, y, t)$	Proportion of values out of range $[-5; 5]$	Values plot
1	4024367	4060553	99.8%	Fig. 45a
0.5	4053105	4016882	99.6%	Fig. 45b
$\frac{1}{2000}$	45	262	$\frac{367}{8100000} = 0.003\%$	Fig. 45c

Table 3: Determining optimal α empirically

Work with data in the form shown in Eq. (1):
The simplest possible model is created to test with data in the form shown in Eq. (1). It is created using the code shown in Fig. 32. The accuracy is well above 0.5 (see Fig. 39) but never exceeds 80% over the epochs. It is worth trying the data processing shown in Eq. (2), which is more complex and interesting as it is based on feedback.

Work with data in the form shown in Eq. (2):
The simple code is changed such that multi-frame videos inputs S_{tot} are now used, with:

$$S_{\text{tot}}(x,y) = \begin{pmatrix} S(x,y,1) \\ S(x,y,2) \\ S(x,y,3) \\ \vdots \\ S(x,y,15) \end{pmatrix} \tag{5}$$

The Python code allowing the creation of the model is given in Fig. 34, and the related performances of this model shown in Fig. 16 are satisfying. As the number of epochs increases, both training and validation accuracies exhibit an upward trend, culminating in a peak validation score of 78%.

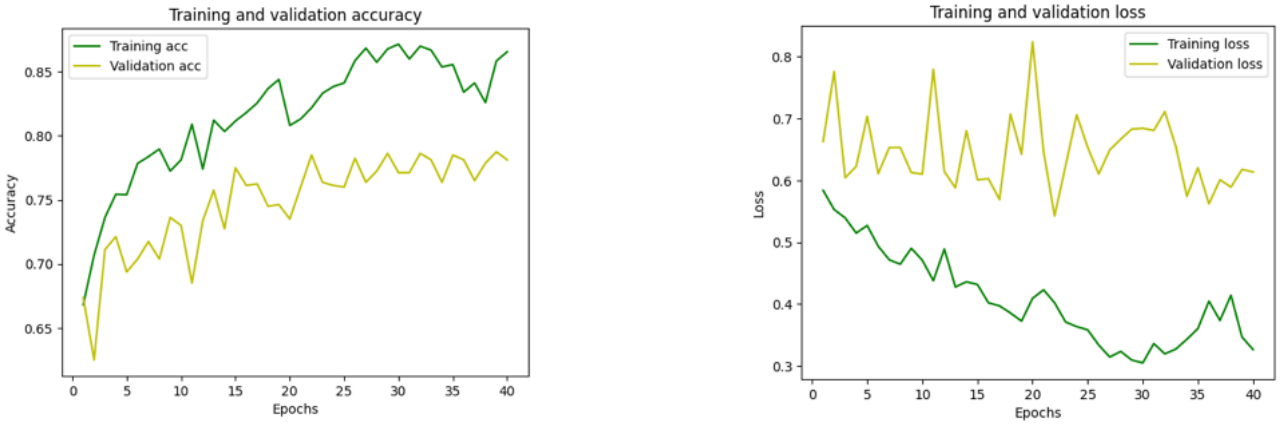


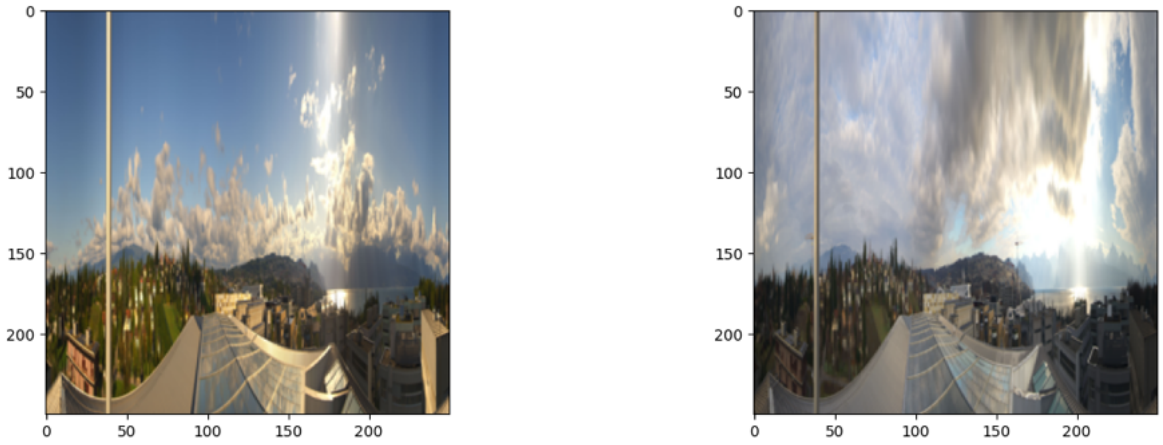
Figure 15: Performances obtained after model simplification by lowering the number of units LSTM

4.2.3 Second approximation - SOLO

SOLO [2] is a function which has been developed within a Python script to simulate, with sufficient accuracy, the output of the optical experiment for a given input.It will be used in the further work.

4.3 Work on new data-set

The new data-set comprises a series of photos captured using a webcam for Intraday Global Horizontal Irradiance (GHI) Prediction. Specifically, the file named `X1.npy` contains a collection of 10,000 RGB images with dimensions $(250 \times 250 \times 3)$. These images were obtained from a camera situated on the campus. The corresponding measured GHI values at the time of image capture are stored in the file `ground_truth.npy`. Additionally, the file `labels.npy` contains GHI values predicted two hours ahead from the time of image capture. In most cases, these predicted values correspond to a forward version (+2 hours) of the GHI values in `ground_truth.npy`.



(a) Image labeled 409.0 (b) Image labeled 31.0

Figure 16: Two random images from the new data-set and their GHI values

In this experiment, the calculation of the Global Horizontal Irradiance (GHI) value 2 hours after the capture of each sky image is attempted, and the comparison of this GHI with those contained in the file `labels.npy` is performed. The accuracy of the prediction is evaluated using the root mean square metric, chosen due to the nature of the labels (floating-point values).

Three different ways of utilizing feedback will be tested and compared to determine which one is the most suitable and yields the best results (refer to Tab. 7 for details about the different concatenation processes). In the subsequent equations, the symbol (+) denotes concatenation rather than algebraic addition.

- **Case 1:** concatenation of cropped images, i.e.:

$$S(t_i + 1) = (1 - \gamma) \cdot \tilde{D}(t_{i+1})) + \gamma \cdot \tilde{f}(D(t_i))^{1} \tag{6}$$

- **Case 2:** concatenation of whole scaled images, i.e.:

$$S(t_i + 1) = (1 - \gamma) \cdot \dot{D}(t_{i+1}) + \gamma \cdot \dot{f}(D(t_i))^{2} \tag{7}$$

- **Case 3:** iterative concatenation of whole scaled images, i.e.:

$$S(t_{i+1}) = (1 - \gamma) \cdot \dot{D}(t_{i+1}) + \gamma \cdot \dot{O}(t_i)) \quad | \quad O(t) = f(S(t)) \tag{8}$$

In these three cases, the augmented data $S(t_i)$ will be provided as input to the classification model. The model was intentionally designed to be as simple as possible and was generated using the Python script depicted in Appendix Figure 33.

[1] The presence of a tilde symbol ($\sim$) indicates that the image has been cropped.
[2] The presence of a dot symbol ($\cdot$) signifies that the image has been rescaled.

4.3.1 Case 1

Method:
This method of utilizing the images is considered the riskiest due to the significant loss of data incurred by cropping the photos. It is not an iterative process since all 5000 output are computed at the same time, such that $f(image(t_i))$ is independent to $f(image(t_{i+1}))$.

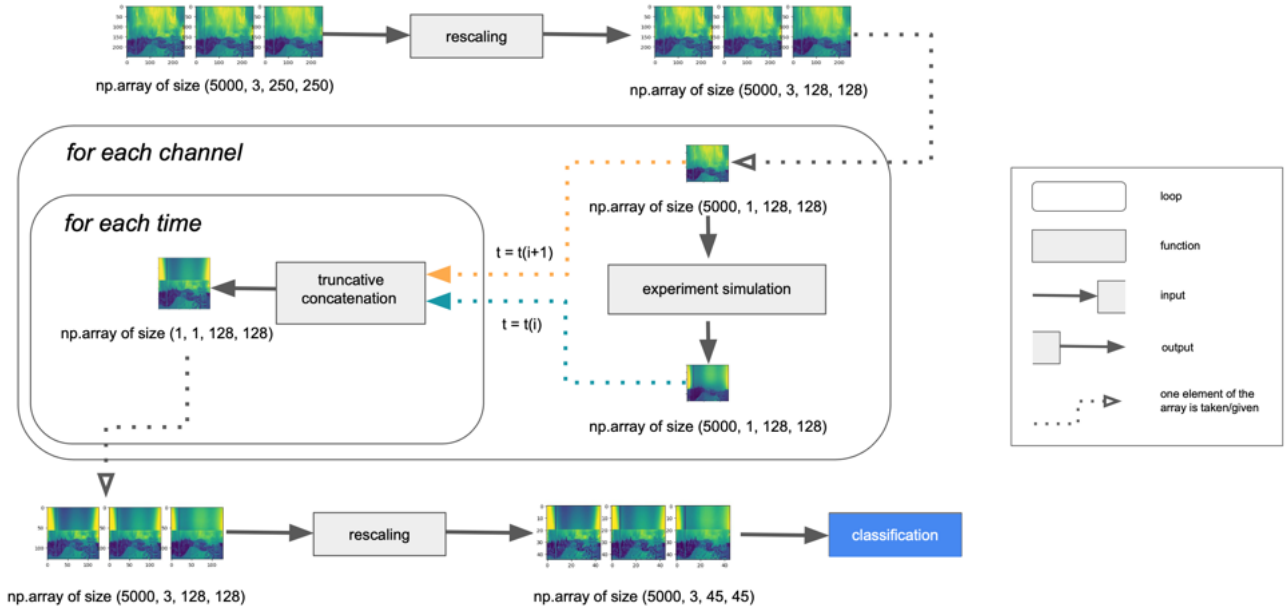


Figure 17: Schematic of Case 1 and its legend - Non-iterative and truncative process

The code utilized for concatenating two images is illustrated in Figure 18. The function `processed_exp()` corresponds to the SOLO function responsible for computing the output of the optical experiment based on a given input array of images. The variable `threshold_pix` was defined earlier in the variable declaration phase and represents the pixel number at which the image is replaced by its output. For instance, for $\gamma = 20\%$, `threshold_pix` $= 26$ because $128 \cdot 20\% = 26$, so 26 out of 128 pixels of $\tilde{f}(D(t_i)$ will injected in $S(t_i + 1)$.

```
for n in range(channels):
    one_channel_output=process_exp(n)
    for i in range(nb_samples-1):
      for j in range(threshold_pix):
        #replace a part of the image by the output
        data[i+1,j,:,n]=one_channel_output[i,j,:]
```

Figure 18: Python code for Case 1

Results:
Using this method, the results for different values of γ are provided in Tab. 4, and the per-

formance graphs are shown in Fig. 40. An effective performance indicator for the models in this case is the best validation accuracy achieved over 40 epochs, represented by the lowest validation root mean square error (or min. val. RMSE).

γ	10%	20%	33%	50%
min. val. RMSE	125	134	129	120

Table 4: Comparison of the performance of Case 1 presented for various values of γ

Discussion:
For the different values of γ, the performance is deemed satisfactory as the label values range from 1.7 to 939.3. Hence, a lower error of 125 corresponds to a maximum relative error of 13% (calculated as 125 divided by the range of label values, 939.3 - 1.7). Furthermore, it is observed that the value of γ does not have a significant impact on the performance in this case.

Practical challenges, such as memory allocation and RAM constraints, emerged during the invocation of the SOLO function. It was not possible to process all the data simultaneously, thus the data had to be fragmented into sub-data or the image quality had to be reduced. Special attention was given to removing all unnecessary np.arrays once they were no longer needed. Tackling these challenges can be necessary, particularly in the context of embedded systems. This practical experience serves as a valuable lesson in overcoming real-world obstacles.

4.3.2 Case 2

Method:
This second case is also non-iterative, but it is expected to be more accurate as the new concatenation method prevents the loss of any part of the data. The Case 2 configuration is illustrated in Fig. 46. In its code implementation (refer to Fig. 19), similar to Case 1, the variable `threshold_pix` is declared and assigned different values based on the γ parameter. Additionally, a function named `scale()` is introduced, which takes an image and the desired image size as input and resizes the given image while applying deformation.

```
smaller_shape = (threshold_pix, 128) #output image size
bigger_shape = (128-threshold_pix, 128) #proper image size

for n in range(channels):
    one_channel_output=process_exp(n)
    for i in range(nb_samples-1):
      data[i+1,(128-threshold_pix):128, :128,n] = scale(one_channel_output
          [i,:,:], smaller_shape[0],smaller_shape[1])
      data[i+1,:(128-threshold_pix), :128,n] = scale(data[i+1,:,:,n],
          bigger_shape[0],bigger_shape[1])
```

Figure 19: Python code for Case 2

In this case, as depicted in the schematic diagram shown in Fig. 46, it is evident that the scaling process occurs only within the *for each time*-loop, indicating that the computation is non-iterative.

Results:
Using this method, the results for different values of γ are provided in Tab. 5, and the performance graphs are shown in Fig. 41.

γ	0%	10%	20%	50%	75%	90%
min. val. RMSE	127	123	125	163	154	149

Table 5: Comparison of the performance of Case 2 presented for various values of γ

Discussion:
It is first observed that the values in Tab. 5 are higher compared to the results in Tab. 4. This indicates that the performance achieved with scaling concatenation is inferior to that obtained with truncative concatenation. Additionally, a significant change in performance is observed between the 20% and 50% γ values. On the left side of the table, the minimum validation RMSE values are around 125, while on the right side, the minimum validation RMSE values are closer to 155. This suggests that the optimal γ value for this particular data-set is below 20%.

4.3.3 Case 3

Method:
This approach represents the most rigorous methodology as it incorporates feedback from the previous image as well as the scaling concatenation. Each output is computed sequentially, and the concatenation step occurs immediately after the output calculation. The corresponding code for this approach is presented in Fig. 20.

```
same= np.empty((1, *new_shape, channels)) #intermediate variable

def process_exp(k,m): #experiment simumation
    one_data= np.empty((1,*new_shape, channels)) #one_data is an array of
        dimension 4 because the function requires the input to be 4
        dimensional
    one_data[0,:,:,m]=data[k,:,:,m]
    inputData=one_data[:,:,:,m]
    outputData=digital_twin.predict(inputData,batch_size = 128)
    one_data=np.squeeze(outputData, axis=-1) #delete the last useless
        demension
    return one_data


for i in range(1,nb_samples-1): #double loop
    for n in range(channels):
      same[:,:,:,n]=process_exp(i,n) #function called for each image
      data[i+1,(128-threshold_pix):128, :128,n] = scale(same[0,:,:,n],
          smaller_shape[0],smaller_shape[1])
      data[i+1,:(128-threshold_pix), :128,n] = scale(data[i+1,:,:,n],
          bigger_shape[0],bigger_shape[1])
```

Figure 20: Python code for Case 3

However, it is important to note that this computational method significantly increases the processing time compared to the other two cases. This is due to the fact that the SOLO function for the experimental simulation is more efficient in computing the output for an entire array of images rather than being called within a loop for each individual image. It is possible that this discrepancy arises from the parallelization of computations when the function has access to multiple images simultaneously. For practical reasons, the reported results are based on a sub-data-set containing only 2000 samples, as opposed to 10,000 samples used in the other two cases.

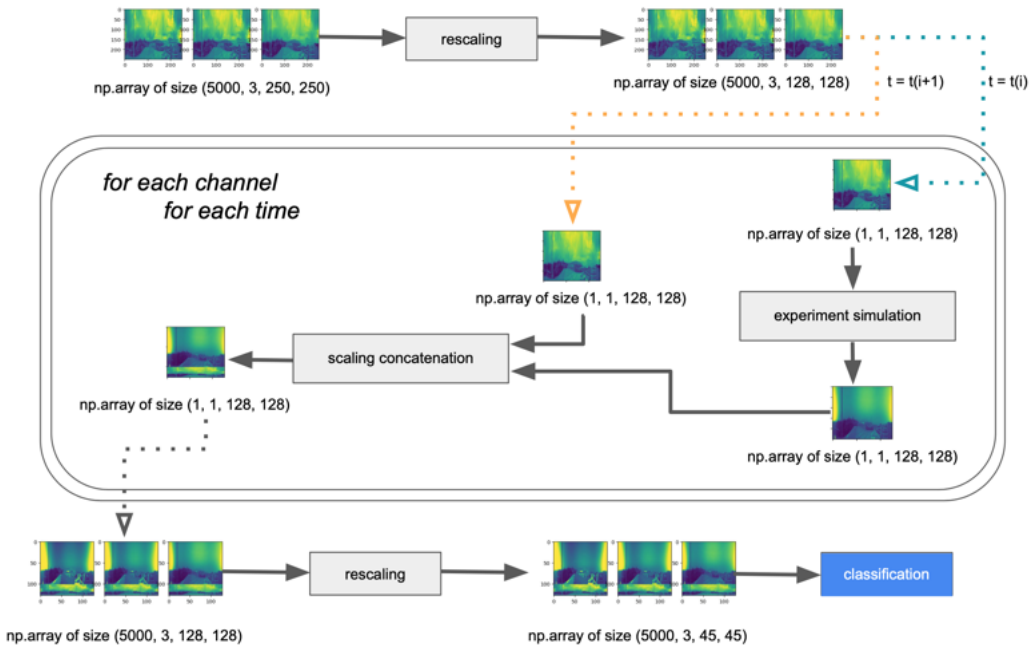


Figure 21: Schematic of Case 3 - Iterative and scaling process

Results:
Using this method, the results for different values of γ are provided in Tab. 6, and the performance graphs are shown in Fig. 42.

γ	0%	10%	20%	33%	50%	90%
min. val. RMSE	122	116	117	121	131	138

Table 6: Comparison of the performance of Case 3 presented for various values of γ

Discussion:
Although the validation accuracies presented in Fig. 42 display a relatively consistent trend across all tested γ values, indicating a limited improvement over the epochs, the results depicted in Tab. 6 reveal outstanding performance specifically for $\gamma = 10\%$ and $\gamma = 20\%$. These two values of γ exhibit minimum root mean square error (RMSE) values of 116 and 117, respectively, which represent the best performance among all three cases. Additionally, considering the label values ranging from 1.7 to 939.3, the lower error of 116 corresponds to a maximum relative error of 12% (computed as 116 divided by the label value range of 939.3 - 1.7). This level of error is deemed sufficient for various applications.

5 Conclusion

In conclusion, the journey through this project has been both challenging and rewarding. The initial exploration with LSTM and GRU neural networks proved to be time-consuming, yet crucial in determining their unsuitability for our data-set. Although the outcomes were not as expected, this experience provided valuable insights into the limitations of these methods. The scores achieved through the utilization of the SOLO algorithm and optical feedback demonstrate comparability (similar magnitude) to what could have been accomplished with complex neural networks. These performances are highly promising, particularly in the case based on an iterative feedback approach.

The subsequent phase of the project, focused on coding work, offered a more fluid progression. The iterative process of experimentation and successful iterations brought about a sense of motivation and accomplishment. It highlighted the significance of hands-on implementation and problem-solving in driving progress.

Moreover, the project provided an opportunity to bridge the gap between theoretical knowledge and practical application in machine learning. Working with neural networks unveiled the power of these tools, captivating my interest and inspiring future exploration. The experience has ignited a passion for using deep learning in personal projects and delving into more advanced classification techniques. It has opened up a vast realm of possibilities for future endeavors. This would not have been possible without the guidance and supervision of Ilker, to whom I express my sincere gratitude.

References

[1] Colah's Blog, Understanding LSTM Networks (2015), https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[2] Uğur Teğin, Mustafa Yıldırım, Ilker Oğuz, Christophe Moser & Demetri Psaltis (2021), *Scalable optical learning operator*

[3] Mushegh Rafayelyan, Jonathan Dong, Yongqi Tan, Florent Krzakala, Sylvain Gigan (2020), *Large-Scale Optical Reservoir Computing for Spatiotemporal Chaotic Systems Prediction*

[4] Che-Lun Hung (2023), *Intelligent Nanotechnology, Deep learning in biomedical informatics*

[5] Nikolay N. Evtikhiev, Vitaly V. Krasnov et al. (2020), *Optical encryption of digital information in the scheme with spatially incoherent illumination based on micromirror light modulators*

[6] Keras, Developer guides (2023), https://keras.io/guides/

[7] Moodle EPFL, Deep learning for optical imaging of Prof. Psaltis (2023), https://moodle.epfl.ch/course/view.php?id=17189

6 Appendix

6.1 Code

```
from keras.applications.vgg16 import VGG16
from keras.models import Model
from keras.layers import Dense, Input
from keras.layers.pooling import GlobalAveragePooling2D
from keras.layers.recurrent import LSTM
from keras.layers.wrappers import TimeDistributed
from keras.optimizers import Nadam

video = Input(shape=(frames,
                     channels,
                     rows,
                     columns))
cnn_base = VGG16(input_shape=(channels,
                              rows,
                              columns),
                 weights="imagenet",
                 include_top=False)
cnn_out = GlobalAveragePooling2D()(cnn_base.output)
cnn = Model(input=cnn_base.input, output=cnn_out)
cnn.trainable = False
encoded_frames = TimeDistributed(cnn)(video)
encoded_sequence = LSTM(256)(encoded_frames)
hidden_layer = Dense(output_dim=1024, activation="relu")(encoded_sequence)
outputs = Dense(output_dim=classes, activation="softmax")(hidden_layer)
model = Model([video], outputs)
optimizer = Nadam(lr=0.002,
                  beta_1=0.9,
                  beta_2=0.999,
                  epsilon=1e-08,
                  schedule_decay=0.004)
model.compile(loss="categorical_crossentropy",
              optimizer=optimizer,
              metrics=["categorical_accuracy"])
```

Figure 22: Python code for VGG-16 CNN and LSTM for Video Classification [?]

```
lab = np.vstack((labels[0, :], labels[1, :]))

plt.xlabel('images')
plt.ylabel('labels')
plt.plot(lab)
```

Figure 23: Python code for printing the labels' repartition

```
num_image = 3000 #or 510 for X_test
fig, axes = plt.subplots(1, 5)
for y in range(0,5):
  axes[y].imshow(X_train[num_image,y,:,:])
  #or axes[y].imshow(X_test[num_image,y,:,:]) for X_test
plt.show()
```

Figure 24: Python code for plotting random X_{test} and X_{train} frames

```
lab = np.vstack((labels[0, :], labels[1, :]))
ohe = OneHotEncoder()
transformed_lab = ohe.fit_transform(lab).toarray()
lab.shape # (4000, 1)
transformed_lab.shape # (4000, 2)
```

Figure 25: Python code for switching into a One-Hot Encoding

```
#data used are the middle images from the video samples

frame_changed=np.ones((4000, 1, 45, 45, 3))
frame_changed=data3[:, 8:9, :, :, :]
data_good_shape = frame_changed.reshape(4000, 45, 45, 3)


# new model without memory
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=
    input_shape))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(units=512, activation="relu"))
model.add(Dense(num_classes, activation ='softmax'))

optimizer = Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
    schedule_decay=0.004)
model.compile(loss="categorical_crossentropy", optimizer=optimizer,
    metrics=["accuracy"])
model.summary()
```

Figure 26: Python code for creating a new model without LSTM

```python
#const definition
frames  = 15
channels= 3
rows    = 45
columns = 45
classes = 2

# model LSTM new try
video = Input(shape=(frames,rows,columns,channels))
cnn_base = VGG16(input_shape=(rows,columns,channels),weights="imagenet",
    include_top=False)
cnn_out = GlobalAveragePooling2D()(cnn_base.output)
cnn = Model(inputs=cnn_base.input, outputs=cnn_out)
cnn.trainable = False

encoded_frames = TimeDistributed(cnn)(video)
encoded_sequence = LSTM(256)(encoded_frames)
hidden_layer = Dense(units=512,activation="relu")(encoded_sequence)
outputs = Dense(units=classes, activation="softmax")(hidden_layer)
model = Model([video], outputs)

optimizer = Nadam(lr=0.002,
                  beta_1=0.9,
                  beta_2=0.999,
                  epsilon=1e-08,
                  schedule_decay=0.004)

model.compile(loss="categorical_crossentropy",
              optimizer=optimizer,
              metrics=["categorical_accuracy"])

model.summary()
```

Figure 27: Python code for creating a new model with LSTM

```
frames=1
num_classes = 2
input_shape = (rows, columns, channels)
video = Input(shape=(frames, rows, columns, channels))

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=
    input_shape))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(units=512, activation="relu"))
model.add(Dense(units=512, activation="relu", kernel_regularizer=l2(0.01))
    ) # regularization
model.add(Dropout(0.5))    # dropout
model.add(Dense(num_classes, activation ='softmax'))

optimizer = Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
    schedule_decay=0.004)
model.compile(loss="categorical_crossentropy", optimizer=optimizer,
    metrics=["accuracy"])
model.summary()
```

Figure 28: Python code for the second try of model without LSTM

```
video = Input(shape=(frames,rows,columns,channels))
cnn_base = VGG16(input_shape=(rows,columns,channels),weights="imagenet",
    include_top=False)
cnn_out = GlobalAveragePooling2D()(cnn_base.output)
cnn = Model(inputs=cnn_base.input, outputs=cnn_out)
cnn.trainable = False

encoded_frames = TimeDistributed(cnn)(video)
#encoded_sequence = LSTM(256)(encoded_frames)
encoded_sequence = LSTM(32)(encoded_frames) #added line
hidden_layer = Dense(units=512,activation="relu")(encoded_sequence)
hidden_layer = Dense(units=512, activation="relu", kernel_regularizer=l2
    (0.01))(encoded_sequence)
hidden_layer = Dropout(0.5)(hidden_layer)
outputs = Dense(units=classes, activation="softmax")(hidden_layer)

model = Model([video], outputs)
```

Figure 29: Python code for adding a LSTM layer on a working CNN

```
video = Input(shape=(frames, rows, columns, channels))
cnn_base = VGG16(input_shape=(rows, columns, channels), weights="imagenet"
    , include_top=False)
cnn_out = GlobalAveragePooling2D()(cnn_base.output)
cnn = Model(inputs=cnn_base.input, outputs=cnn_out)
cnn.trainable = False

encoded_frames = TimeDistributed(cnn)(video)
encoded_sequence = GRU(256)(encoded_frames)
hidden_layer = Dense(units=512, activation="relu")(encoded_sequence)
outputs = Dense(units=classes, activation="softmax")(hidden_layer)
model = Model([video], outputs)
```

Figure 30: Python code for creating a model based on GRU

```
input_shape = (frames, rows, columns, channels)
video = Input(shape=input_shape)

cnn_base_input = Input(shape=(rows, columns, channels)) #base
cnn_base = Conv2D(filters=32, kernel_size=(3,3), activation='relu')(
    cnn_base_input)
cnn_base = MaxPooling2D(pool_size=(2, 2))(cnn_base)
cnn_base = Conv2D(filters=64, kernel_size=(3,3), activation='relu')(
    cnn_base)
cnn_base = MaxPooling2D(pool_size=(2, 2))(cnn_base)
cnn_base = Conv2D(filters=128, kernel_size=(3,3), activation='relu')(
    cnn_base)
cnn_base = MaxPooling2D(pool_size=(2, 2))(cnn_base)

cnn_out = GlobalAveragePooling2D()(cnn_base) #output
cnn = Model(inputs=cnn_base_input, outputs=cnn_out)
cnn.trainable = False
encoded_frames = TimeDistributed(cnn)(video)
encoded_sequence = GRU(256)(encoded_frames) #gru layer
hidden_layer = Dense(units=512, activation="relu")(encoded_sequence)
hidden_layer = Dense(units=512, activation="relu")(hidden_layer)

outputs = Dense(units=classes, activation="softmax")(hidden_layer)

model = Model(inputs=[video], outputs=outputs)
```

Figure 31: Python code for creating a model based on GRU with trainable convolutional layers instead of VGG as convolutional preprocessing layers

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(2025,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(2, activation='softmax'))
```

Figure 32: Python code creating of a simple model based on a few dense layers

```
const=0.9*nb_samples
#get the right sizes for X_train, X_test... according to nb_samples

X_train = data_final_ds[:int(const)]
y_train = cut_labels[:int(const)]
X_test = data_final_ds[int(const):]
y_test = cut_labels[int(const):]
input_shape = (45, 45, channels)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=
    input_shape))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='linear'))

optimizer = Nadam(lr=0.002, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
    schedule_decay=0.004)
model.compile(loss="mean_squared_error", optimizer=optimizer, metrics=["
    RootMeanSquaredError"])
model.summary()

print("Fit model on training data")
history = model.fit(
     np.asarray(X_train),
     np.asarray(y_train),
     batch_size=64,
     epochs=40,
     validation_data=(np.asarray(X_test), np.asarray(y_test)),
)
```

Figure 33: Python code creating of a simple model for the 3 cases

```
input_shape = (frames-1, rows, columns)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=
    input_shape))
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(2, activation='softmax'))
```

Figure 34: Python code creating of a simple model based on a few dense layers, for data of another shape

6.2 Models performances

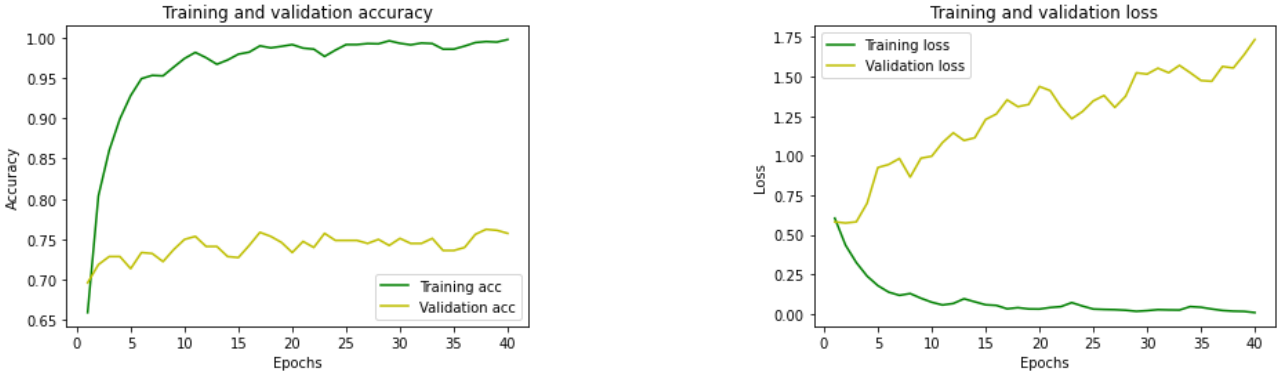


Figure 35: Performances obtained after model simplification by lowering the number of units LSTM

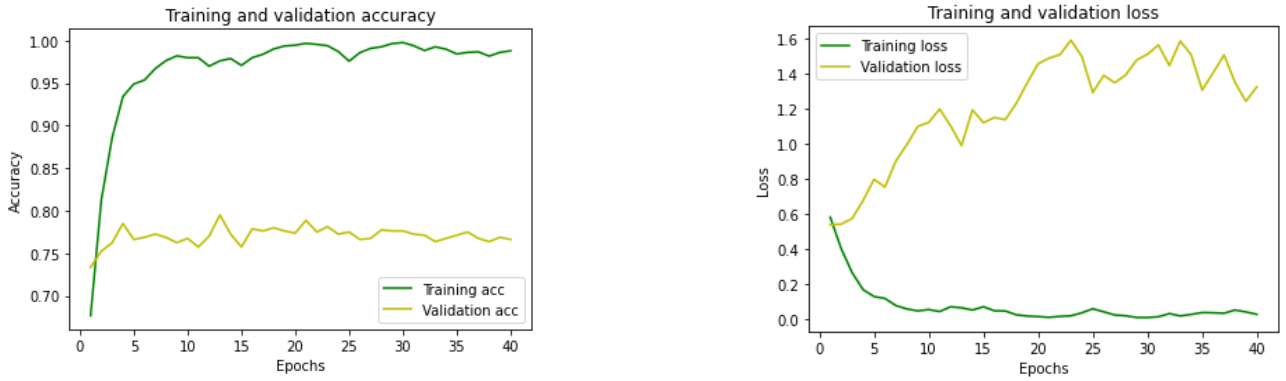


Figure 36: Performances obtained after model simplification by lowering the number of images per video

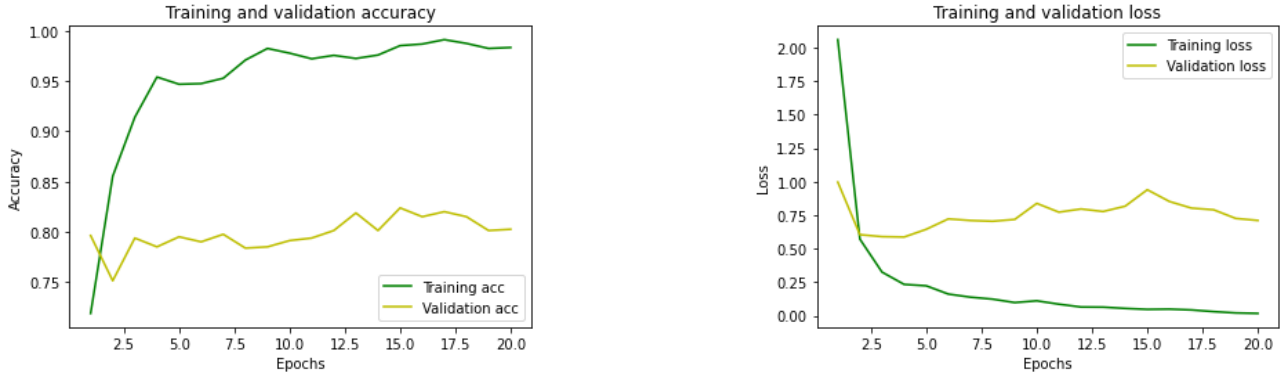


Figure 37: Performances obtained after adding regularization to the dense layers

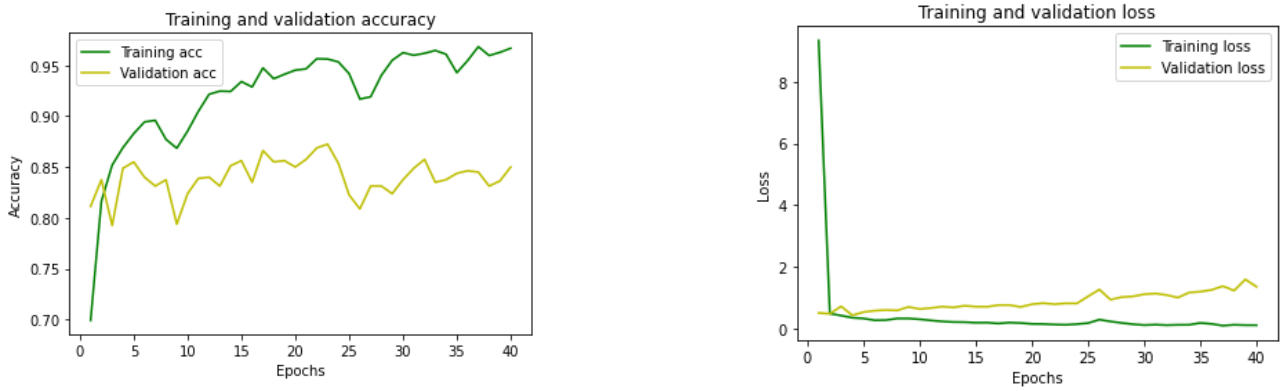


Figure 38: Performances obtained after removing the LSTM layers

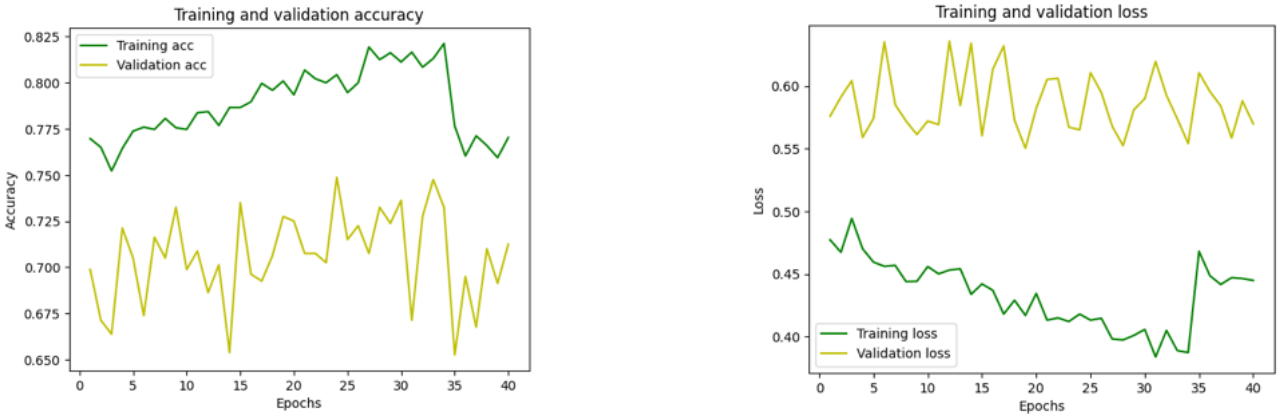


Figure 39: Performances obtained with a simple model based on a few dense layers for the data in the form shown in Eq. (1)

6.3 Other

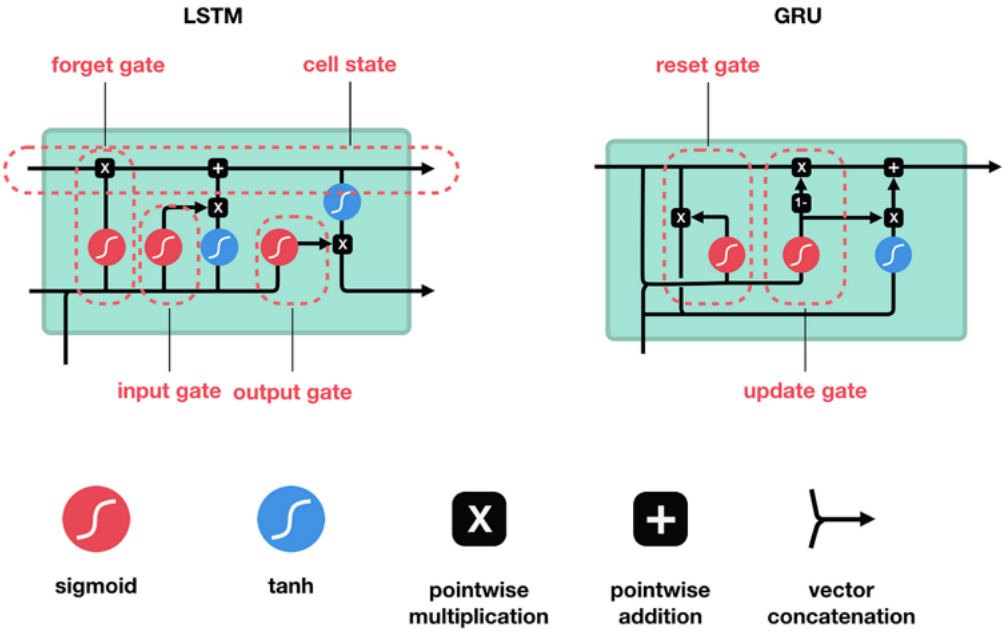
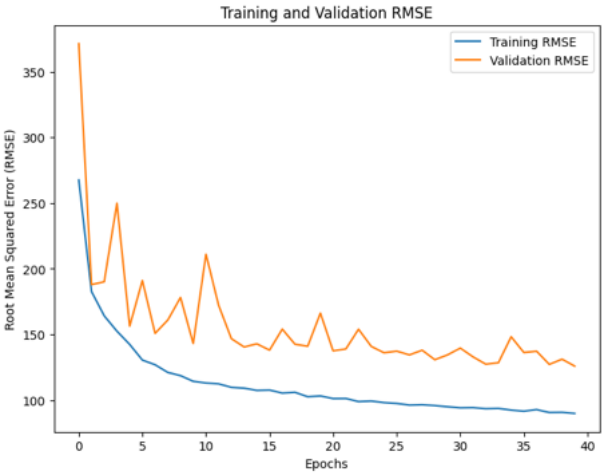
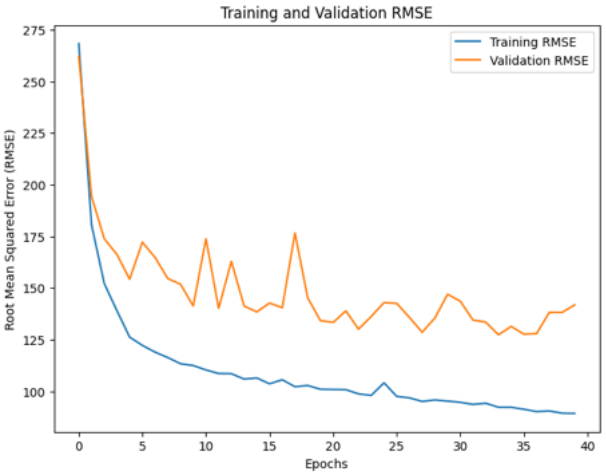


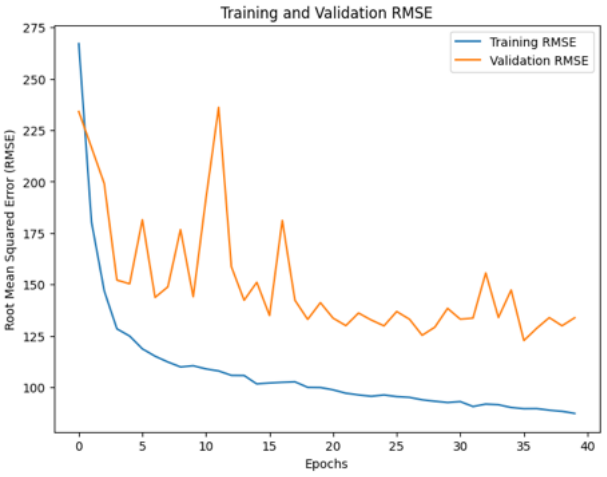
Figure 43: Comparison of a LSTM cell and a GRU one



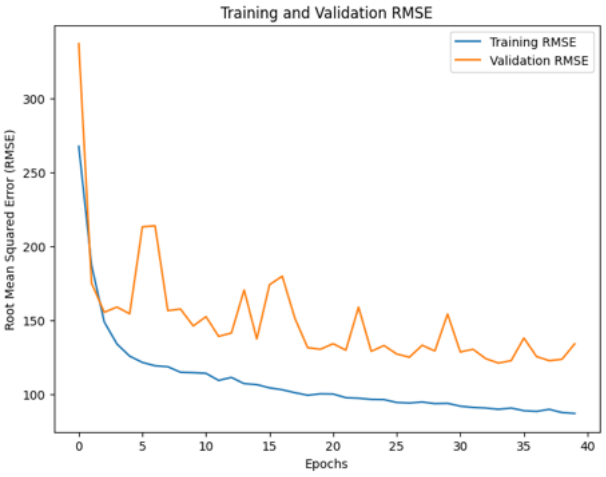
(a) $\gamma = 10\%$



(b) $\gamma = 20\%$

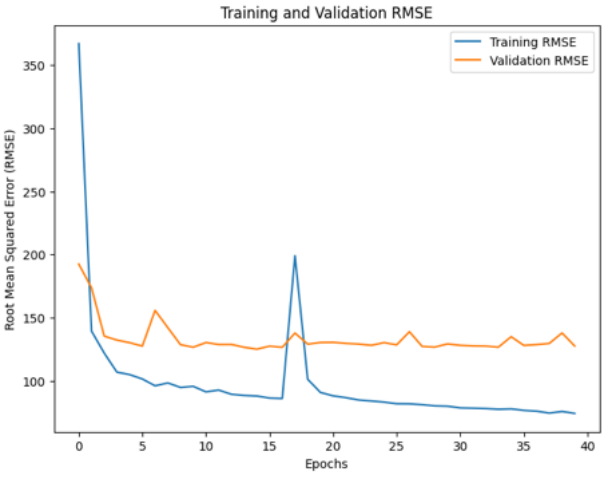


(c) $\gamma = 33\%$

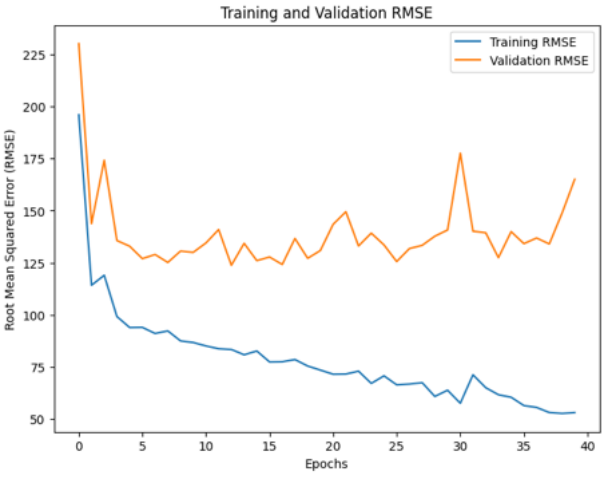


(d) $\gamma = 50\%$

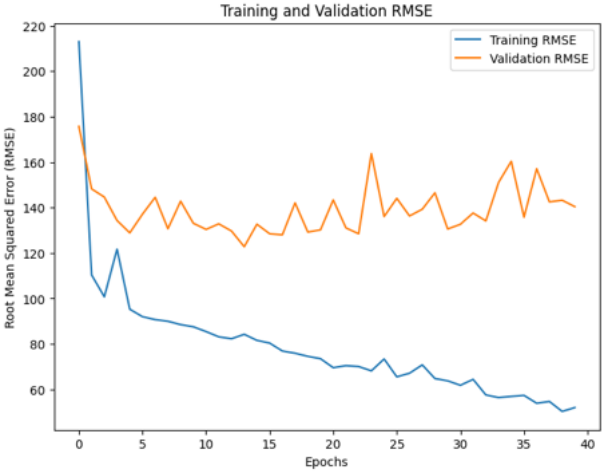
Figure 40: Performances of Case 1 for different γ values



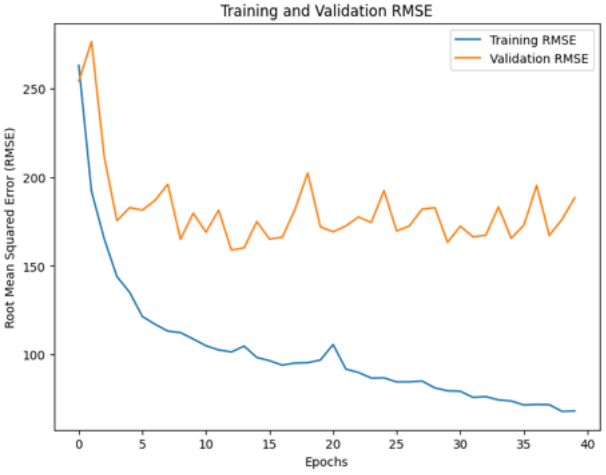
(a) $\gamma = 0\%$



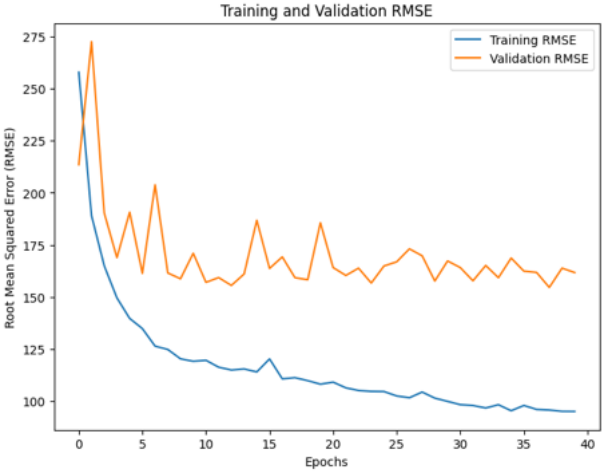
(b) $\gamma = 10\%$



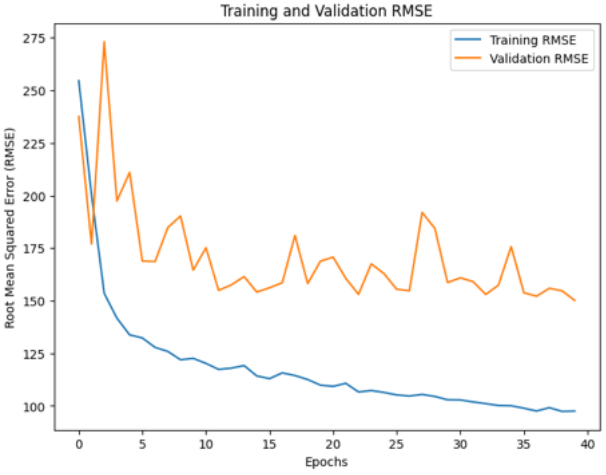
(c) $\gamma = 20\%$



(d) $\gamma = 50\%$



(e) $\gamma = 75\%$

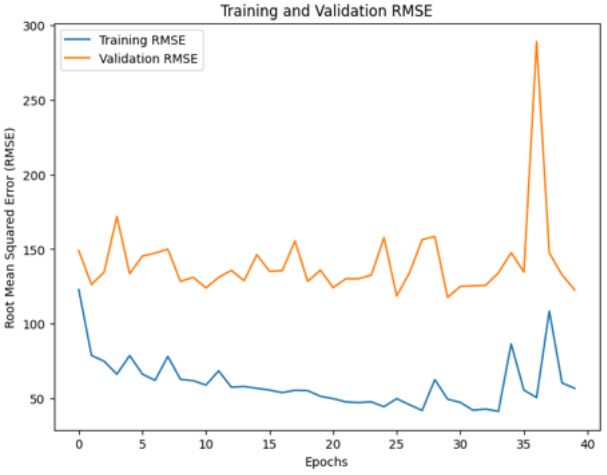


(f) $\gamma = 90\%$

Figure 41: Performances of Case 2 for different γ values

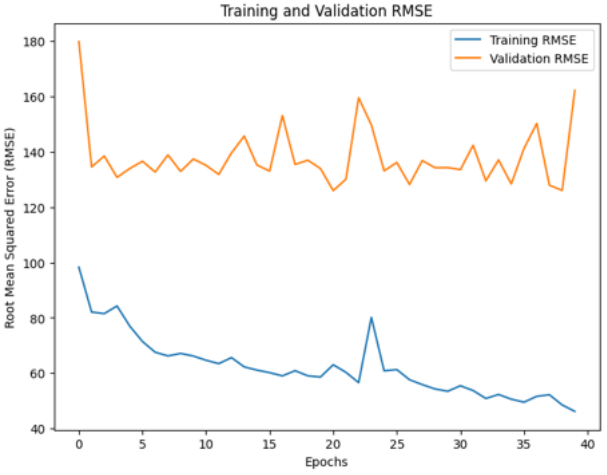
(a) $\gamma = 0\%$

(b) $\gamma = 10\%$

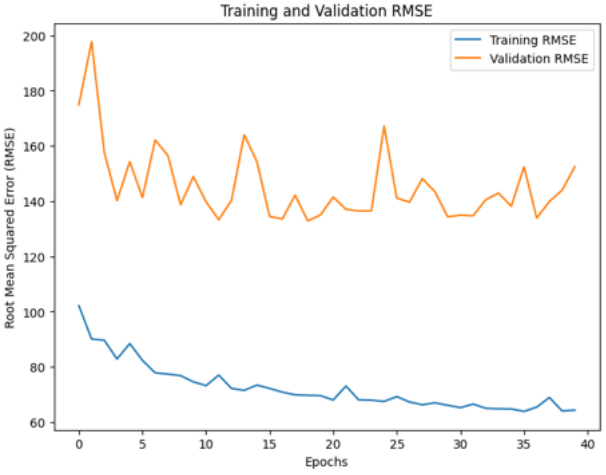


(c) $\gamma = 20\%$

(d) $\gamma = 33\%$



(e) $\gamma = 50\%$



(f) $\gamma = 90\%$

Figure 42: Performances of Case 3 for different γ values

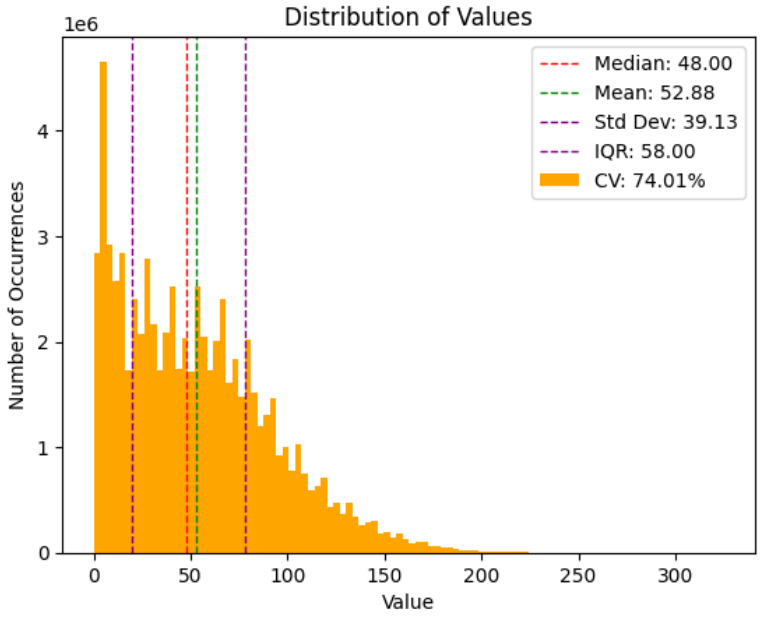
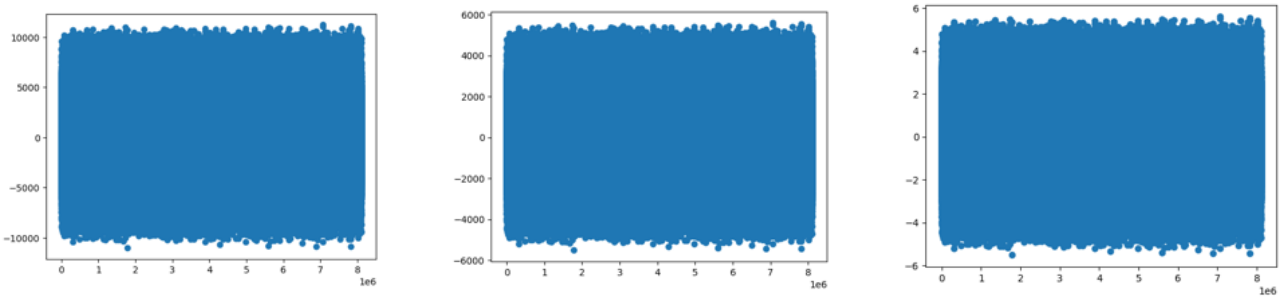


Figure 44: Distribution of data values and indicators for inspection



(a) $\alpha = 1$ (b) $\alpha = 0.5$ (c) $\alpha = \frac{1}{2000}$

Figure 45: Plots of $\alpha \cdot M \times S(x, y, t)$ values for various α values

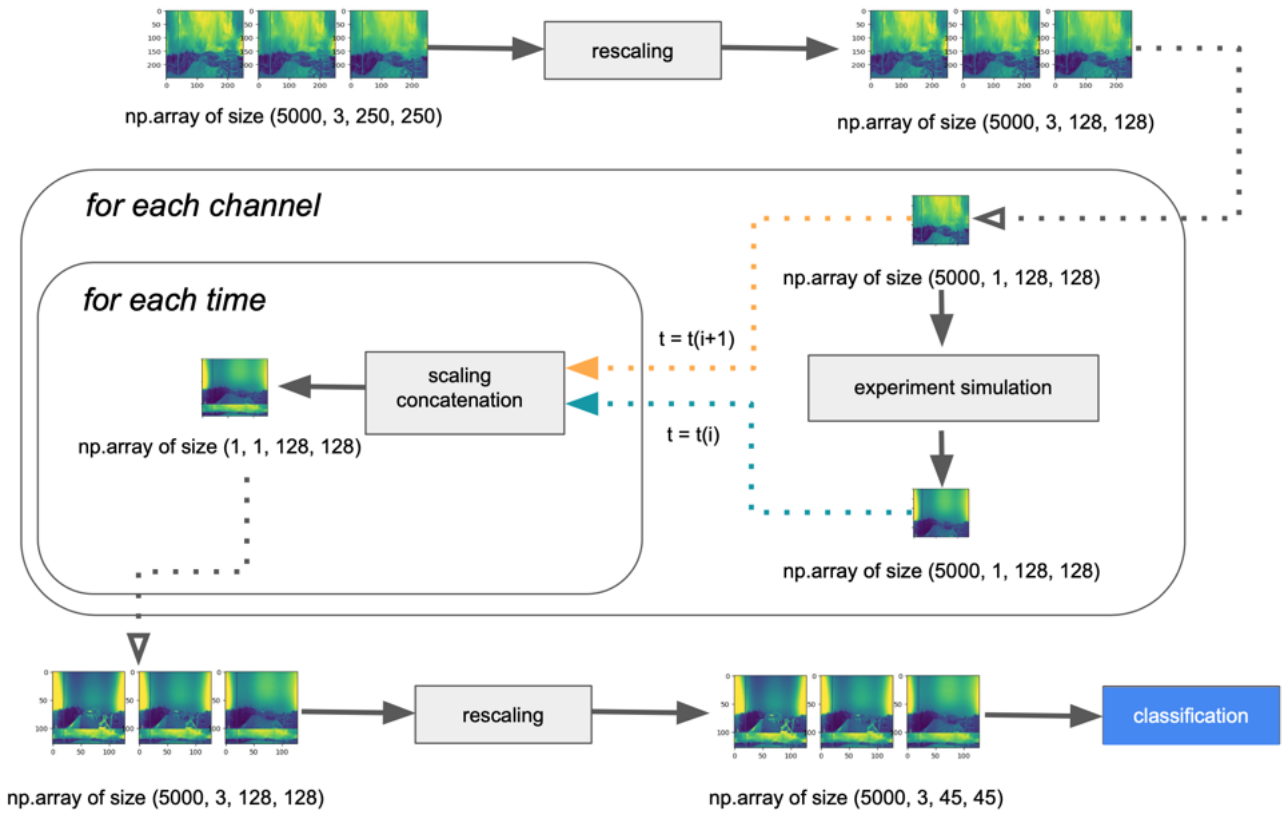


Figure 46: Schematic of Case 2 - Non-iterative and scaling process

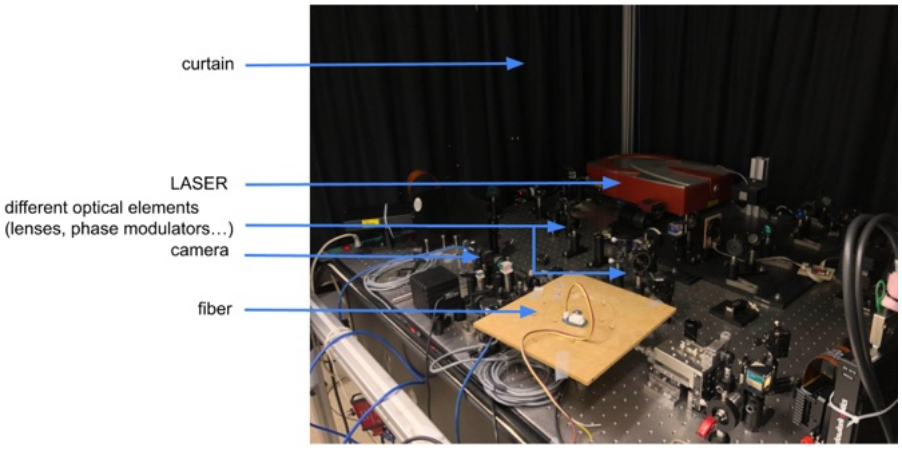


Figure 47: Experimental set-up of the SOLO exeperiment

Image 1	Image 2
	
Truncative concatenation	**Scaling concatenation**
	

Table 7: Graphical representation of truncative and scaling concatenation