Reconstruction of a vorticity field behind a cylinder

Cécile Maguin, Charlotte Durand , Simone Le Chevère

May 18th, 2020

Introduction

In this project, we aim to reconstruct the vorticity field of a fluid flow behind a circular cylinder, from the measurements taken by a few sensors positioned on the back of the cylinder (where the vorticity field appears).

Our objective is to reproduce the entire vorticity field on a length which is ten times the radius of the cylinder, from information taken at the surface of the cylinder only. For that purpose, we use a shallow deep learning model. By modifying the architecture and parameters of our algorithm, we try to optimize the accuracy of the model and the time of the training.

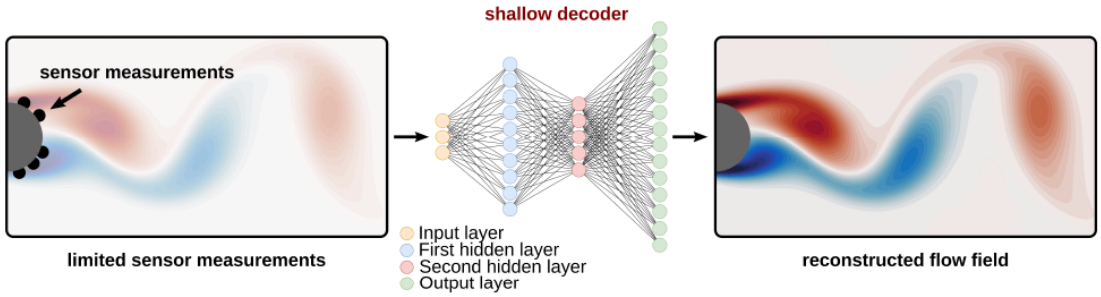


Figure 1: Schematic of the shallow decoder presented in the article.

This project is inspired by this article : Erichson, N. & Mathelin, Lionel & Yao, Zhewei & Brunton, Steven & Mahoney, Michael & Kutz, J.. (2019). Shallow Learning for Fluid Flow Reconstruction with Limited Sensors and Limited Data.

We used the same data set as this article. First, we proposed to recreate a fully connected layers model, like in the article, using two hidden linear layers. Then, we introduced a new model which uses a deconvolution layer, hoping to accelerate the training process and lightening memory usage. This report presents the comparison of the two models, and also offers several analysis on the influence of different parameters on the training efficiency.

1 Presentation of the code

The main program is a Jupyter notebook called "Vorticity_field_project.ipynb". The main program creates the two types of models described in this report, trains them over a same random set of 5 sensors, and plots their results. Models are defined in an annex python file called "models.py", and utility functions are defined in a python file "utils.py".
Below we present the utility functions the program uses.

1.1 Data set and sensors

The data set is composed of 150 images with a size of 384x199. 384 is the dimension along the vertical axis(X) and 199 is the size along the horizontal axis(Y). The cylinder is centered at : X=0 and Y=99 (only half is represented), and has a radius of 36 pixels. The 150 images are a temporal sampling of the flow which is running at the back of the cylinder.

From this data, we will extract "sensors measurements" by taking the vorticity field values of the pixels corresponding to the defined sensors positions. The position of a sensor on the cylinder is defined by its angle θ relative to the x-axis of the image. The utility function **define_sensors_coordinates_from_theta** converts any array of θ positions into x and y pixel-coordinates.

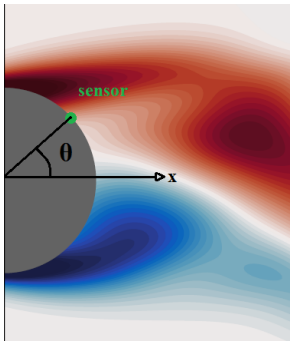


Figure 2: Definition of the angle θ of sensor positioning.

Additionally, utils.py provides the function **random_sensors_positionning**, which returns a random array of n_sensors distinct couples of coordinates, separated by at least 2° on the cylinder's surface. To be able to have reproducible results, **random_sensors_positionning** takes also into argument the desired random seed to be used for the random python generator.

In utils.py, you will find additionally a function called **data_treatment**, which transforms the raw data into suitable data for the training sequence. It splits the data set into two parts : the first part is the training set (Training_data) which contains a number Training_size of images; and the remaining images consist of the validation set (Validation_data). The function also creates inputs arrays corresponding to sensors measurements for both training and validation sets. All these arrays are converted to tensors for Pytorch to handle, and images are linearized into 1D to

be adapted to the model processing. This data is then normalized by the absolute maximum of values of the training set plus an $\epsilon = 0.02$, so that no value of the final training set is equal to 1. This norm is recorded to be later used for reshaping data as required. The model will thus work with data normalized between -1 and 1.

1.2 Model training sequence

utils.py also provides a function **train_model** to train any model over a number of epochs specified by the value n_epochs. The training proposed is supervised online minibatch training. For each epoch, the model is trained by minibatches of examples taken in a random order from the training set. The loss function is calculated through Mean Square Error (MSE Loss) of generated examples compared to the true examples of the training set. The optimizer is an Adam optimizer, which changes adequately the learning rate over the training. Besides, the loss function calculated by the Adam optimizer adds a term equals to the weight decay parameter times the norm of the matrix of weigths, in order to not have unreasonable growth of weights over the learning phase.

At each epoch, the loss value is recorded, and an error is calculated on the validation set by averaging over a minibatch of validation examples the root squared error on each pixels. If p_{ij}^k is the generated value of pixel ij for the validation example k, and $\hat{p}_{ij}^k$ is the true value of the pixel, then the error calculated is :

$$\text{Error on validation minibatch} = \frac{1}{\text{minibatch size}} \sum_{k \in \text{mininbatch examples}} \sqrt{\frac{\sum_{i,j}(p_{ij}^k - \hat{p}_{ij}^k)^2}{\sum_{i,j} \hat{p}_{ij}^{k\,2}}}$$

The function thus takes into argument the model, the data (input for training and validation, and expected outputs for training and validation, and training parameters (initial learning rate, the weigth decay, the size of the minibatch and the number of epochs). It returns the trained model, the loss function and the error evolution during training, and additionally the total computing time.

2 Presentation of the two models

2.1 Model with fully connected linear layers

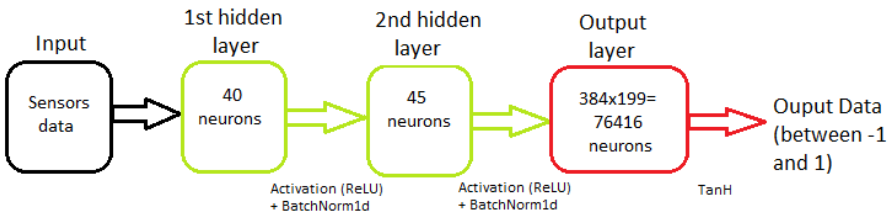


Figure 3: Schematic description of the FCLL model.

We first propose a model imitating the one provided by the reference article. The model consists of two hidden linear layers, and an output linear layer. Because all neurons in weighted linear layers are connected with each other, we will nickname this model the fully connected linear layers model (FCLL).

The FCLL model consists of two hidden layers, of sizes 40 and 45, which are activated by a ReLU function, and are followed by a gaussian normalization with the module BatchNorm1d. The output layer is a linear layer followed by Tanh as an output function, and provides 384x199 generated pixels of value between -1 and 1, to generate the image corresponding to the sensors measurements.

2.2 Introduction of a model with a deconvolution layer

The main time and memory consuming step of the fully connected linear layers model, is the last linear layer, which has a huge number of neurons: $199 \times 384 = 76416$ neurons. In order to reduce the load of this crude layer, we propose in this section to introduce a deconvolution layer. Because the vorticity field values are continuous, adjacent pixels have related values. This is why a deconvolution layer could prove to give good results, by sort of "dezooming" the image as the last layer of the model. The final image is thus the image obtained from the first three linear layers, but with a "super resolution".

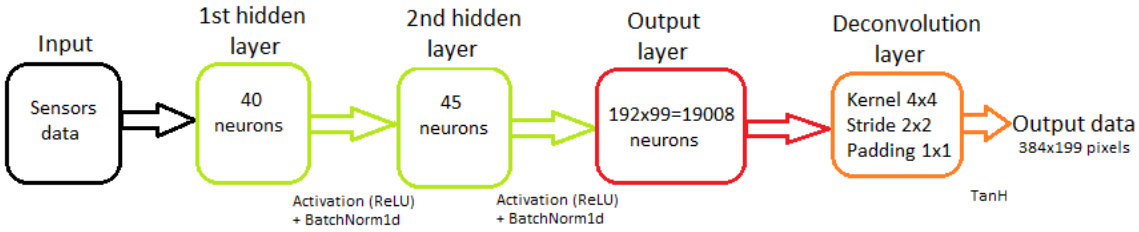


Figure 4: Schematic description of the deconvolution model.

The deconvolution layer is constructed through the **ConvTranspose2d** module of Pytorch. We impose a stride of 2 by 2, to dezoom approximately by 2 the image. We add a padding of 1 by 1, to be able to compute properly the borders of the image. We choose to have a square kernel, which means that the kernel has the same size for the x dimension and the y dimension. Figure 5 depicts the kind of deconvolution we want to achieve.

The height H_{out} and width W_{out} of the image in pixels after the layer **ConvTranspose2d** is calculated by :

$$H_{out} = (H_{in} - 1) \times \text{stride}[0] - 2 \times \text{padding}[0] + \text{dilation} \times (\text{kernel size} - 1) + \text{output padding}[0] + 1$$

$$W_{out} = (W_{in} - 1) \times \text{stride}[1] - 2 \times \text{padding}[1] + \text{dilation} \times (\text{kernel size} - 1) + \text{output padding}[1] + 1$$

We keep the dilation to 1 (default). Output padding is an option to add a line of 0 on the desired dimension. We will use this option to have the right dimension of the image as output. Because

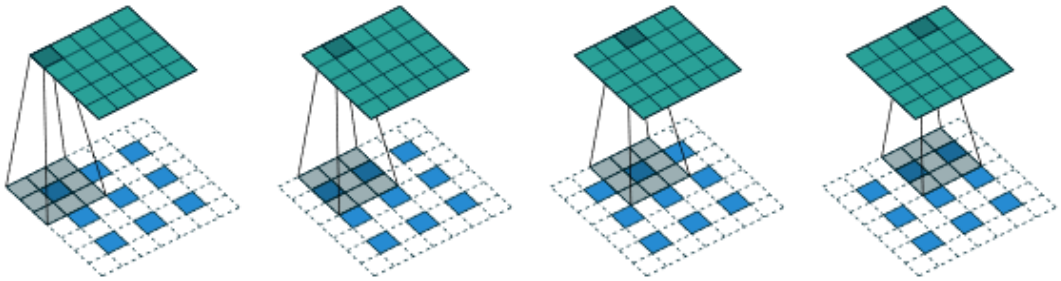


Figure 5: Depiction of the transpose of convolution of a 3x3 kernel over a 3x3 input, padded by 1x1, with 2x2 stride. Blue map is input, cyan map is output. Source : `https://github.com/vdumoulin/conv_arithmetic`.

H_{out} and W_{out} are fixed by the dimension of the images of the data set, we will calculate the dimension H_{in} and W_{in} that should be obtained as output of the first three linear layers :

$$H_{in} = \frac{H_{out} - \text{kernel size} - \text{output padding } [0] + 2 \times \text{padding}[0]}{\text{stride}[0]} + 1$$

$$W_{in} = \frac{W_{out} - \text{kernel size} - \text{output padding } [1] + 2 \times \text{padding}[1]}{\text{stride}[1]} + 1$$

Output padding is thus adjusted to have integer values for H_{in} and W_{in}. In our program, the kernel size is by default 4x4, but can be changed when defining the model. In the rest of the report, we will consider a 4x4 kernel size if not expressed otherwise.

2.3 Comparison of the two models

In this section, we compare the performances of this deconvolution model with the previous fully connected linear layers (FCLL) model. We use the same random sensor generation for each model to compare them properly. A first overview of results is provided in figures 6. Both models provide decent reproduction of vorticity field, even if the deconvolution seems a little bit more accurate. Looking at the loss function and the error on validation set during training (figure 7), we see that that the deconvolution model trains faster relatively to the number of epochs. However, the computation time of the deconvolution model is approximately 1.5 times higher than the FCLL model.

To get a more accurate and quantitative comparison, we train both types of models on 10 different random generated set of 5 sensors, during 2500 epochs. Results are averaged over the 10 runs, and are presented in the boxplots in figure 8. Especially, we define the convergence time of a model as the time it needs to get to its final value of loss or error $\pm$ 5%. Overall, the deconvolution model offers better results, with lower values of loss and error. It has faster convergence speed in terms of epochs, approximately 3 times faster than the FCLL model. But it still needs $\times 1,5$ more time to compute the same amount of epochs (Note that the computation times provided here are relative to the computing machine used, and are not relevant for other computing devices). Thus, taking into account the convergence speed and the computation speed of both models, we

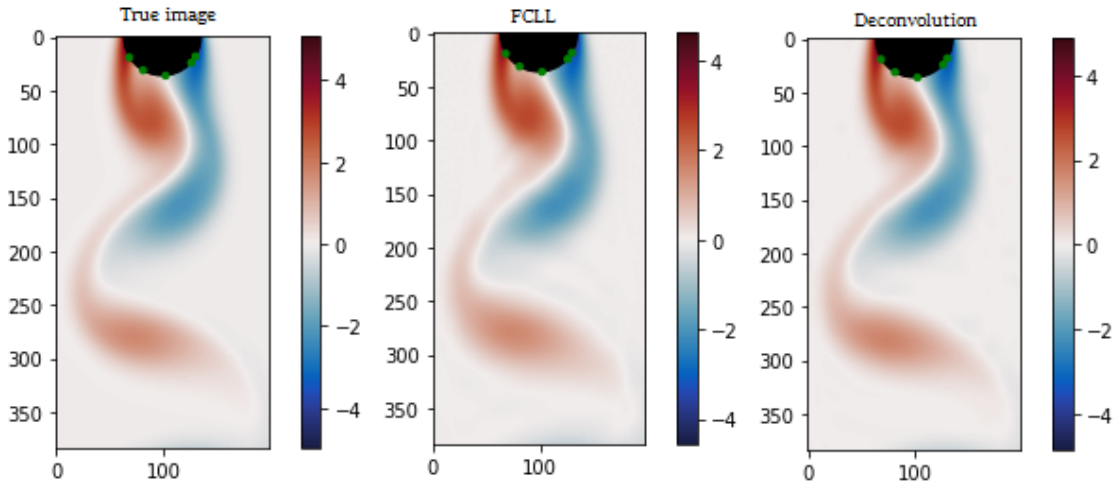


Figure 6: Vorticity field images (26th example); sensors are represented in green. On the left, the image from the data set; in the middle the image generated by the FCLL model after training during 4000 epochs; on the right the image generated by the deconvolution model after training on the same set of sensors during 4000 epochs.

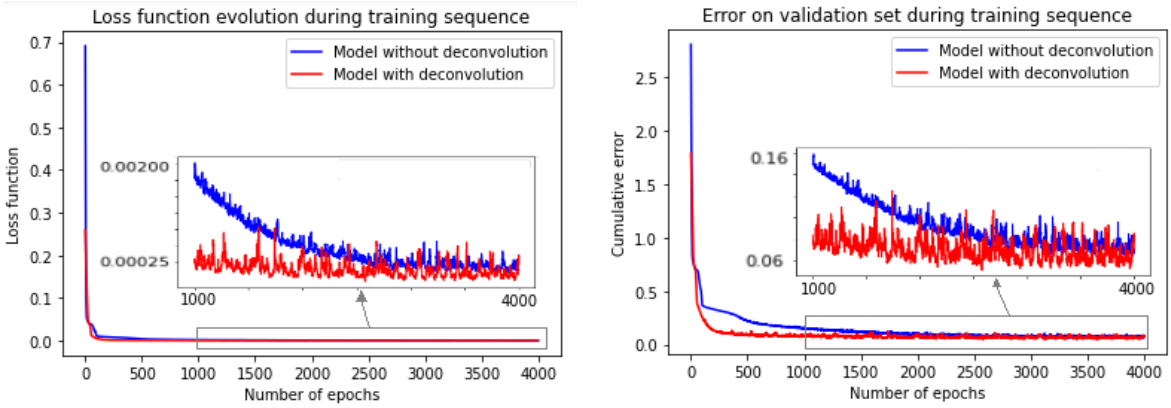


Figure 7: On the left, evolution of loss during training for both models; on the right, evolution of the error on the validation set for both models. Boxes in grey offer a zoom between 1000th and 4000th epochs. These results were obtained for 5 sensors generated with random seed #3.

can conclude that by using the deconvolution model (and properly adjusting the number of epochs needed), training can approximately be 2 times faster.

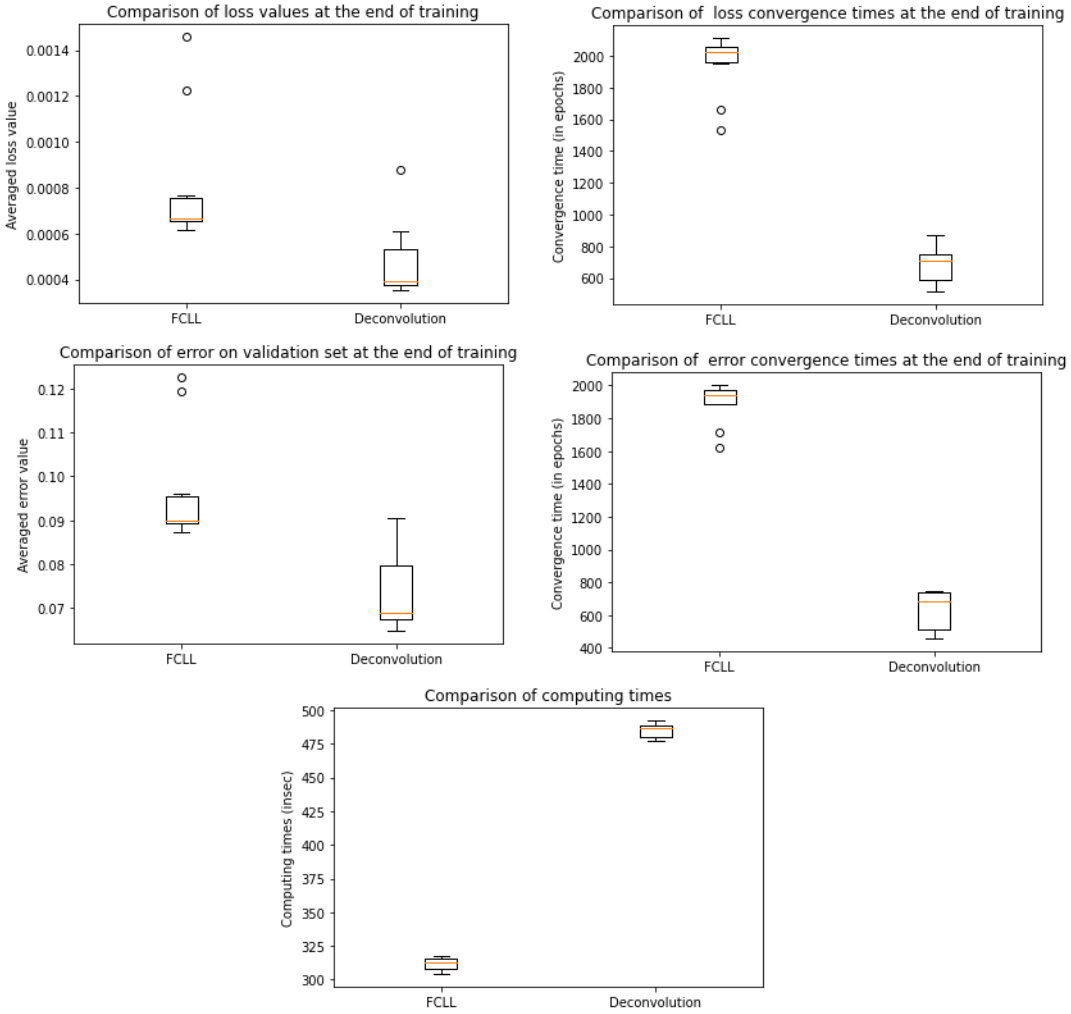


Figure 8: Statistically averaged results over 10 different random sets of sensors for the FCLL model and the deconvolution model. Final values of loss and error are averaged over the last 200 epochs. Convergence times correspond to the epoch where the loss/error reaches its final value +5%. Computing times (in seconds) are relevant to the computer which trained the models.

3 Analysis of the influence of different parameters

In this sections, we provide additionally analysis on the influence over training of several parameters, such as model parameters (activation function, kernel size), training parameter (minibatch size) and sensors measurements (number of sensors and their positions).

3.1 Choice of the activation function

We wanted to know the influence of the activation function for the two first layers, for our program and our dataset. In this part, we chose four different activation function : ReLU, Sigmoid, SoftPlus and SoftShrink.
Those four activation functions are plotted in the Fig. 9.

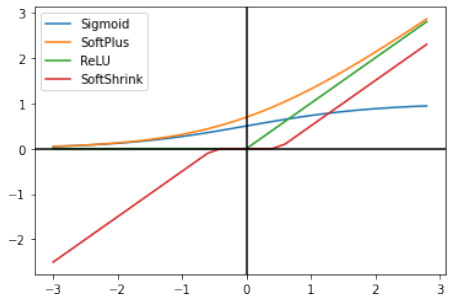
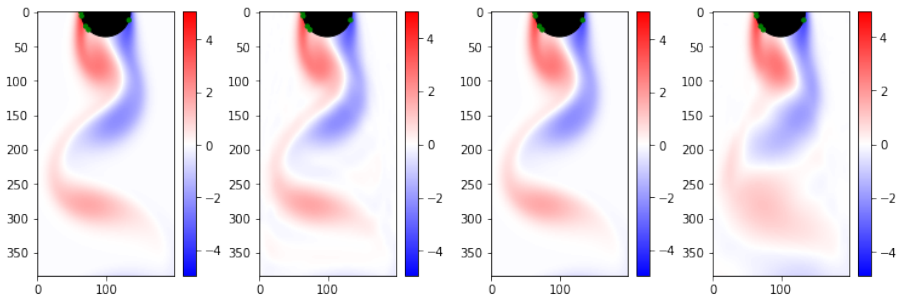
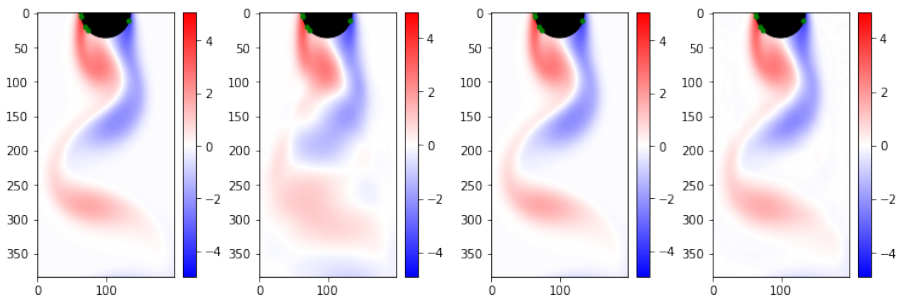


Figure 9: 4 Activation Functions studied



(a) ReLU

(b) Sigmoid



(c) SoftPlus

(d) Softshrink

Figure 10: Reconstructed flow field for different activation functions. For each activation function, we have on the left the real flow, and on the right the reconstructed flow

In Fig. 10, we can observe the influence of the activation functions for the reconstructed flow with always the same parameters : 2000 epochs, 7 sensors with the same position, and a 0.001 learning rate. Apparently, the Sigmoid function and the SoftPlus function don't give good results. Both ReLU and SoftShrink functions seem to give acceptable results.
In the Fig. 11, we have much more information on the influence of the four activation functions. The two top figures show that those activation functions don't have an extremely important influence. However, when we zoom on the first 500 epochs, we can observe that, as the reconstructed flow showed, both Relu and SoftShrink have much better results than the Sigmoid and SoftShrink, on both error validation and loss function. On the bottom of the graph, we can observe the statistics of the last 1000 epochs : ReLU offers the best results : it has the smallest number of outliers : it offers a better stability. This is why the ReLU is used as an activation function for the remaining tests.

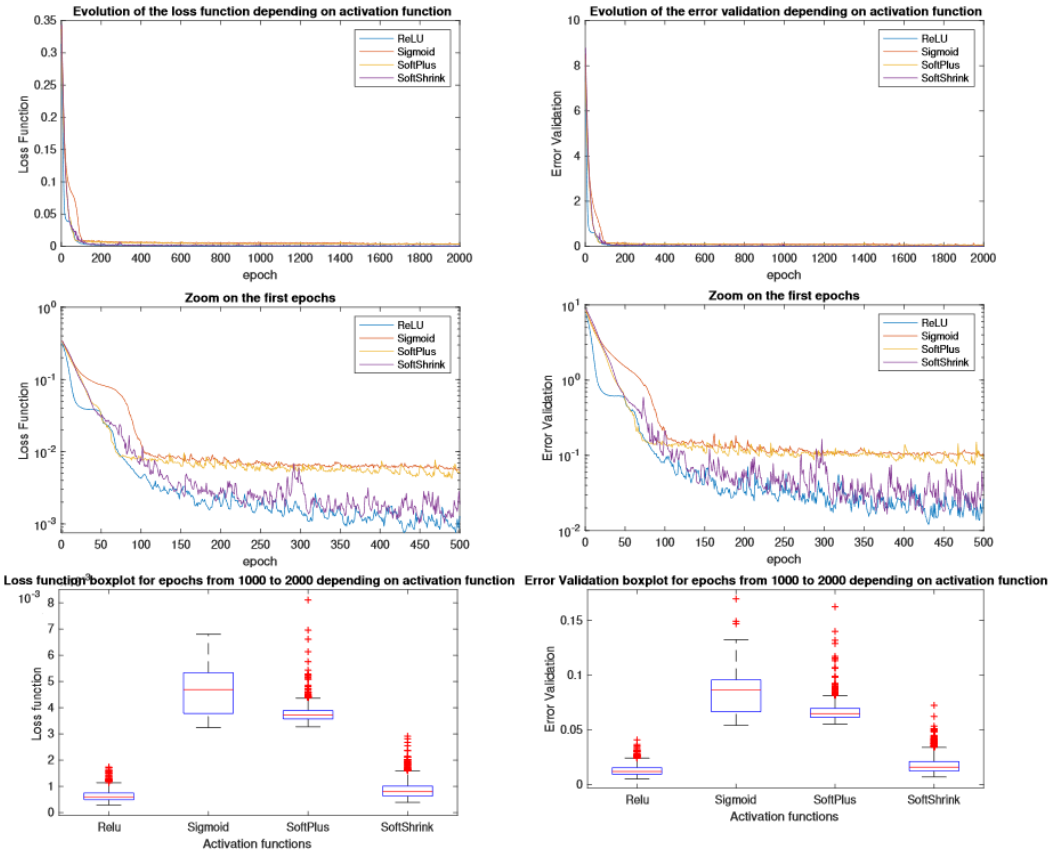


Figure 11: (top) : Evolution of the loss function and the error validation for different activation functions on the 2000 epochs.(Middle) : Zoom of the evolution of the loss function and the error validation on the first 500 epochs. (Bottom) : Loss function and error validation statistics on the last 1000 epochs. Red cross represents outliers.

3.2 Influence of kernel size

We arbitrarily decided to operate the deconvolution layer with a 4 by 4 kernel. However, we can wonder if the results would be better with another size of kernel. For 10 different sets of 5 sensors, we tested kernel sizes from 2 by 2 to 7 by 7 by training over 2000 epochs (see figure 12). Overall, when the kernel is bigger, the accuracy of the model at the end of training is better, which is logical because pixels of the image are approximated from more data. The variance of the results also decreases when kernel size increases, which indicates more stable results. However, computation times are notably higher for bigger kernels. Convergence times are similar for any kernel size.

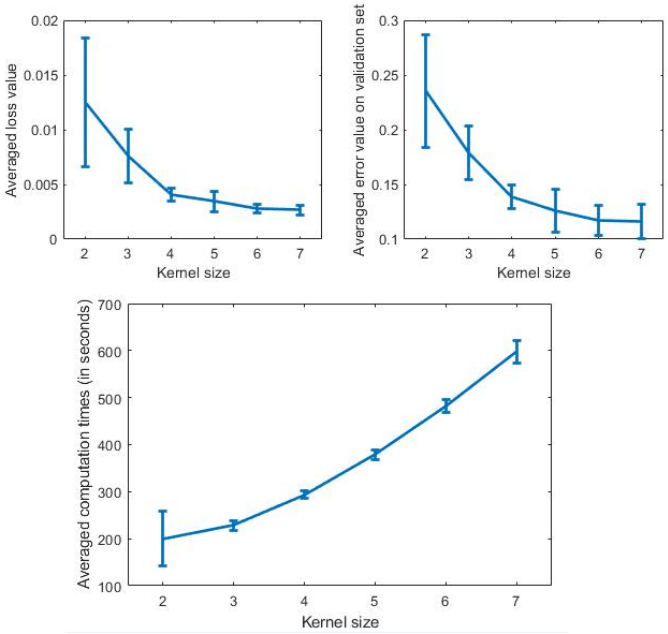


Figure 12: Averaged results over 10 runs of deconvolution model training over 2000 epochs for different kernel sizes. Above, the final loss value and the final error on validation set averaged for the last 200 epochs and for the 10 runs. Below, averaged computation times for the 2000 epochs training. Error bars represent standard sample deviation calculated over the 10 results.

In conclusion, the kernel size should be chosen as a compromise between the accuracy we want for the model and the computing time needed.

3.3 Influence of minibatch size

In this part, we want to observe the influence of the size of the minibatch on the algorithm. Results can be observed in Fig. 13. We chose a 0.001 learning rate, 7 sensors with same positions and 2000 epochs. We can immediately see that the smaller the minibatch size is, the bigger the error validation is (top right and middle left). Also, it looks like there is much more noise on the smaller mini-batch size. This is also seen on the two bottom graphs : there is an obvious decrease of both error validation and loss function with the increase of the mini-batch size. Especially, there are

many outliers with mini-batch from size 5 to size 20 : it means there is more noise for those sizes. Both size 25 and 30 seem to offer good results, with excellent error validation and small noise. Also, on the middle right graph, we can observe the calculation time depending on the mini-batch size. The calculation time is significantly decreasing with the mini-batch size, which is not intuitive since there is much more calculation with big mini-batch size.
For the others studies, we chose a 30 mini-batch size.

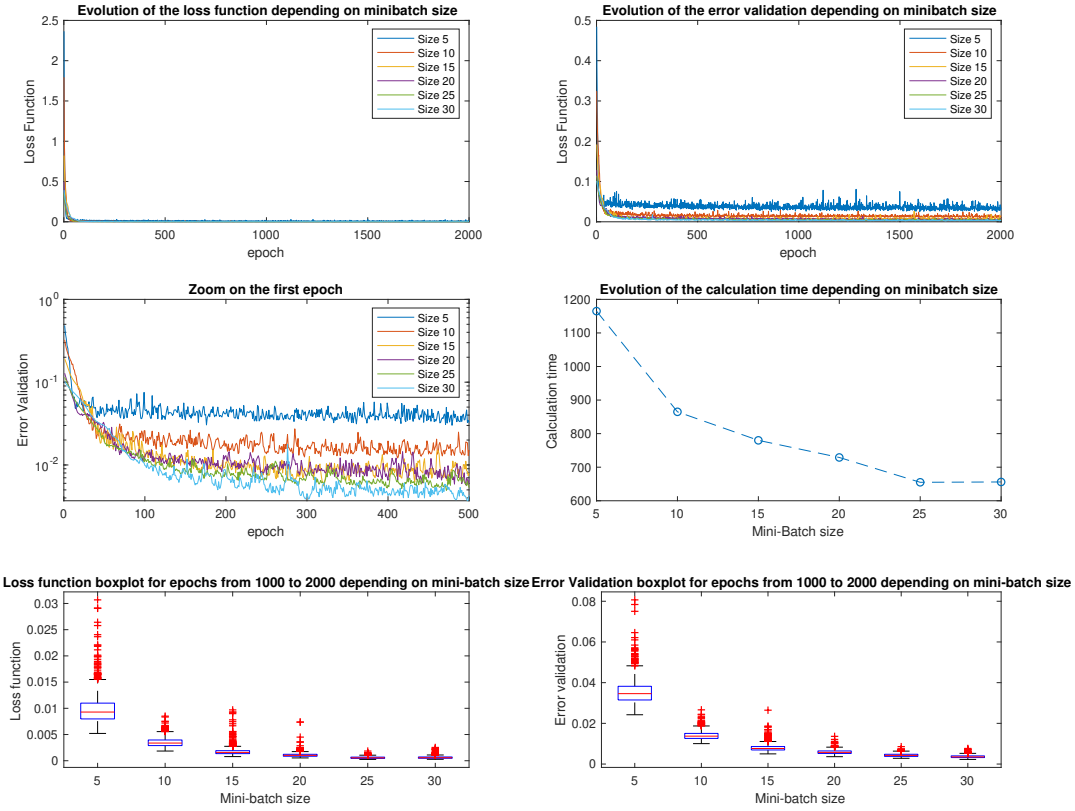
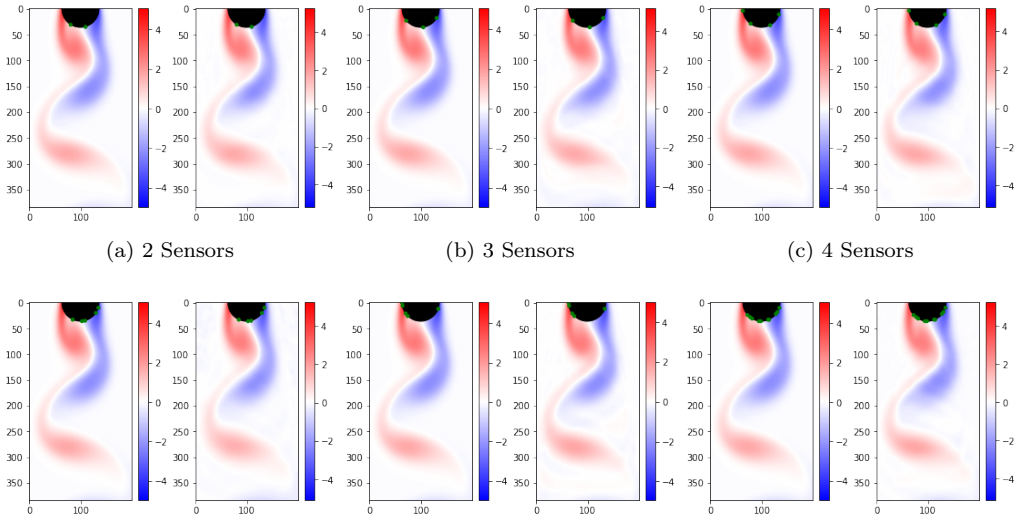


Figure 13: (top) : Evolution of the loss function and the error validation for different minibatch size on the 2000 epochs.
(Middle) : Zoom of the evolution of the error validation on the first 500 epochs (left) and evolution of calculation time (right) for different minibatch size.
(Bottom) : Loss function and error validation statistics on the last 1000 epochs. Red cross represents outliers.

3.4 Influence of the number of sensors

In this section, we studied the influence of the number of sensors used to predict the flow. For each number of sensors, we made 5 runs for better reproductibility. All the runs are made on 2000 epochs and with a 0.001 learning rate.
In the Fig. 14, we can observe the reconstructed flow depending on the number of sensors. First of all, we can see that the algorithm is able to reconstruct the flow for any number of sensors, even for two sensors. We can sometimes see some defaults on the reconstruction, but it is not that important. To see clearly the influence of the number of sensors, we need the Fig. 15.
On the first four graphs, we can observe the statistics of the influence of the number of sensors on the last 1000 epochs for all the runs (5000 points for each number of sensors). Clearly, for 2 and 3 sensors, results aren't good : both loss function and error validation are much higher than for more sensors. Furthermore, there are many outliers, which mean there is a lot of noise. For more than 4 sensors, we can see that the loss function and the error validation are globally constant. The only difference is that with the increase of the number of sensors, we reduce the noise on both variables. The next 3 graphs show, as we have seen before, that the algorithm is able to learn for any number of sensors. Yet, when we zoom on the first 500 epochs, we can see that there is a difference between 2 and 3 sensors, and more than 4 sensors. Finally, with the last graph, we can look at the calculation time depending on the number of sensors. It seems we reach a maximal time after 4 sensors. Adding more sensors do not increase the calculation time.

To conclude, for a consistent analysis of the algorithm, we need to chose more than 4 sensors. The more sensors we select, the less noise there will be, however we will not decrease the error validation.



(a) 2 Sensors (b) 3 Sensors (c) 4 Sensors

(d) 5 Sensors (e) 7 Sensors (f) 10 Sensors

Figure 14: Reconstructed flow field for different numbers of sensors. For each number of sensors, we have on the left the real flow, and on the right the reconstructed flow

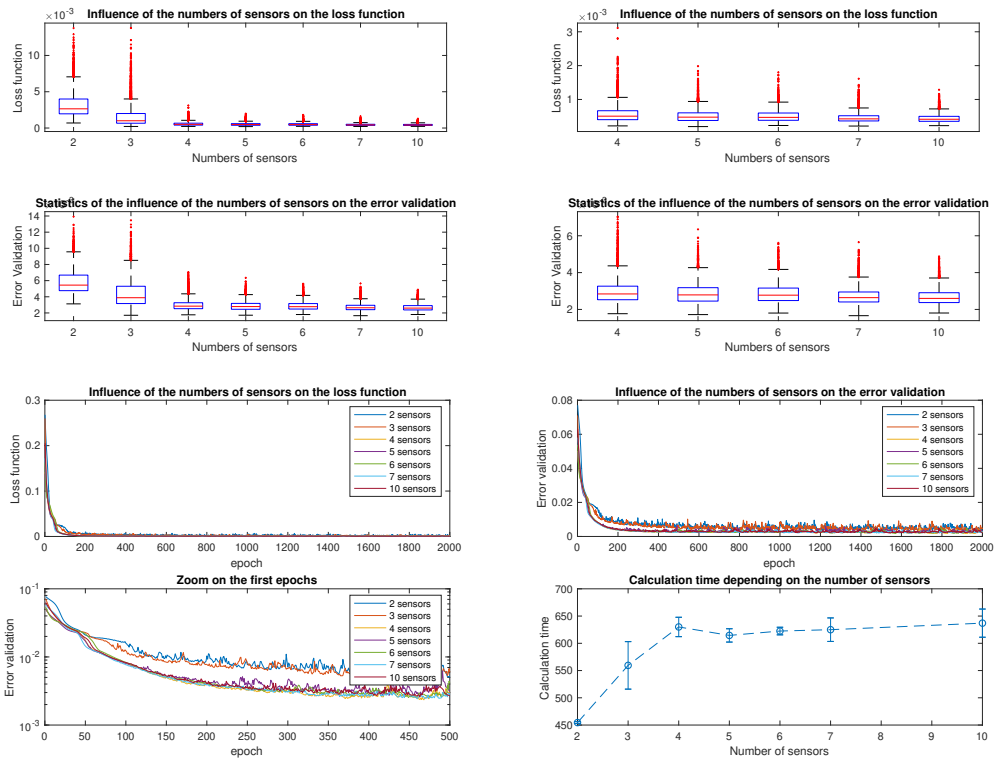


Figure 15: (top and below) : Statistics on 5 runs for the last 1000 epochs on error validation and loss function. Red cross represents outliers
(Middle) : Evolution of the loss function and the error validation on 2000 epochs for one run of each number of sensors for more visibility.
(Bottom) : Left : Zoom on error validation on the first 500 epochs. Right : Influence of number of sensors on calculation time.

3.5 Choice of sensors positions

As we saw before, our model is able to compute even when there are only 2 sensors, although the variance of results is high in this case. One can thus wonder if the positions of sensors on the cylinder have an influence on the ability of the model to train. To investigate the effect of the position of sensors, we tried to train the model with only 2 sensors with chosen positions defined by their angle θ on the cylinder. We varied the combinations of θ_1 and θ_2, with angles from -80° to +80° by 10° increment. Results were averaged over 2 runs of 1000 epochs. It could be noted that anti-symmetrical positions were not problematic for the model to learn. However, positions who were the most problematic, and gave the highest error values at the end of training, seemed to be the ones where the sensors were especially close to the x axis (that is, θ_1 and θ_2 close to 0, as seen

in figure 16).
This is consistent with fluid mechanics theories, because at the back of the cylinder placed in a fluid flow, there is a detachment of the turbulent boundary layer. We can suppose that these positions thus give less information for the model to learn. The error value observed is thus slightly higher.

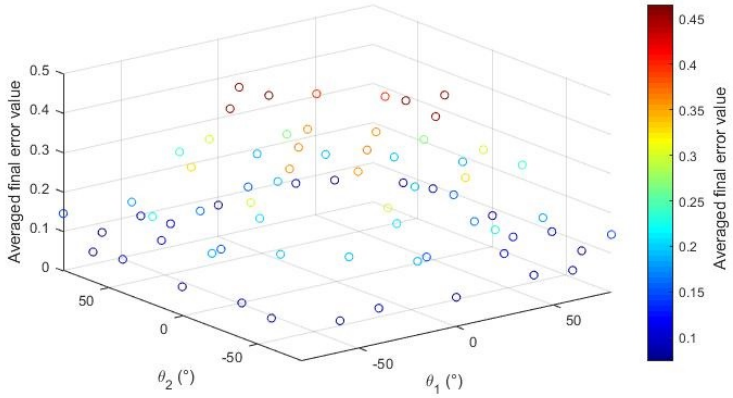


Figure 16: 3D scatter plots of the averaged error values at the end of training over 1000 epochs, for the deconvolution model, when only 2 sensors are selected. The positions of the 2 sensors are defined respectively by θ_1 and θ_2. Values of error seems to be higher when θ_1 and/or θ_2 are close to 0.

4 Conclusion

In this project, we succeeded in creating a neural network, using minibatch training, which computes accurately the vorticity field behind a cylinder with sensors measurements. Compared to previous articles, we added a deconvolution layer which enhances the convergence speed of the model and lowers the loss and error values.

By running our program with different parameters, we were able to determine the best choice for the activation function, and some criterias to choose the size of the kernel, the size of the minibatch, and finally the influence of the number and positions of the sensors. To guarantee the robustness of our results, we took the average of a series of runs for each parameter. In summary, to get the best results, one should take a large size of minibatch, and a large kernel size for the model, with ReLU as the activation function. Measurements should be done with at least 4 sensors. However, if it is not possible to have that many sensors in real-life experiments, sensors should preferably be placed on the side of the cylinder and not directly on the back of the cylinder (opposed to the flow, close to the x axis).
To go further, one could wonder the influence of the two linear layers' sizes on the loss and error values and the running time. Some of our measurements indicate that their size do not have a great influence on those parameters. By computing the error for different sizes between 20 and 10.000, we did not observe a certain tendency and the error did not exceed 0.1 in every case. However, we did not have time to average these results over multiple runs, so we cannot conclude with certainty.