# Characterization of turbulent flows in tokamaks

Julien Hu, Matthieu Masouyé and Sébastien Ollquist
*Department of Computer Science, EPFL, Switzerland*

***Abstract*—The goal of this project was to use machine learning to analyze GPI measures and estimate the Z-axis velocity of the structures.**

**To do that, we have used a convolutional neural network (CNN) on a resnet 3d architecture, with the help of a synthetic dataset we generated ourselves.**

**We were able to successfully train and optimize a CNN for each dataset, and overall we were able to have good estimation of the speeds; however, time constraints prevented us from going as far as we would have liked.**

## I. Introduction

A tokamak is a type of plasma generator, inside which plasma turns at very high speeds. The plasma is confined in the shape of a torus via magnetic fields. At the edges of this field, it separates from the flow in a turbulent fashion; this is called the shear layer. These turbulences might create "blobs" of plasma, that can be ejected at high speed.

Gas Puff Imaging (GPI) is a technique used to study these turbulences: it consists in launching a puff of neutral gas near the shear layer, to generate light emissions when mixing up with the plasma and thus allowing imaging of the turbulent structures. These images are cross-sections of the plasma, with two axis: an $r$-axis perpendicular to the wall of the torus, and a $z$-axis going vertically and parallel from the $r$-axis, along side the shear layer.

GPI analysis is an ongoing task, and can yield important results. The Swiss Plasma Center lab at EPFL was interested in the new possibilities machine learning could bring, and together we worked to test how well a neural network could analyze these images.

The primary objective was to measure the $z$-axis velocities of the moving, turbulent structures captured by GPI. We quickly decided to go for a convolutional neural network working on sequences of frames, as it seems a good approach for complex tasks and apt for image sequence analysis and velocity estimation [1].

## II. Data and preprocessing

### A. Real Data Description

The labeled data was given to us from the lab in pickle files, inside which there are the following sets:

1) `shot` the shot number.
2) `t_window` an array of timestamps at which the frames were measured, in seconds
3) `brt_arr` a 3D-array (of 12x10x$l$ where $l$ is the length of `t_window`) of G.P.I. measures, corresponding to the brightness of each 12 by 10 frames, measured in $mW/cm^2/ster$
4) `r_arr` the array of $r$-coordinate of each view, in meters.
5) `z_arr` the similar array of $z$-coordinate of each view, also in meters.
6) `vz_from_wk` the vertical speed for each column of data, in $km/s$.
7) `vz_err_from_wk` the error margin in the estimation of vz
8) `col_r` the average $r$-coordinate of each column.

Due to the measurement method, four specific "pixels" of `brt_arr` are set to NaN permanently. The `r_arr` and `z_arr` describe the exact position of each pixel of `brt_arr` in a 2d space, as they are not perfectly spaced. `vz_from_wk` will constitute the labels for the algorithm.

Importantly, there are 13 values in `vz_from_wk`, `vz_err_from_wk` and `col_r` while there are only 12 columns per frames: this is because the shear layer we want to calculate is inside one of the columns, which means this column will have flows going both ways. Thus, this column will have two speeds associated to it. This also means that `col_r` has two identical values, as they concern the same column.

Note that it wasn't possible for the lab to provide good estimates of these two speeds in all datasets, and the 13th column will be set to NaN if they couldn't get an accurate estimation. Additionnally, the direction of the plasma relative to the shear layer is consistent: the left side always has plasma going down (i.e. negative values), while the right side always going up (i.e. positive values).

### B. Synthetic Data Description

We have decided to generate artificial data mainly for two reasons: first is that the real dataset would take some time for the lab to prepare; and more importantly it would give us a controlled environment with no noise or imprecision in the labels, so that we could use this dataset to evaluate different architectures.

With the help of the lab directors, we could generate this in the following way: we draw a large canvas (by default 480x480), and spawn in random gaussian arrays of varying small sizes (around 264 arrays of 100 to 260 pixels wide). A ratio of these arrays are also set to negative values. To draw a frame, all the arrays are summed in the canvas, and values are limited to between 0 and 255 to stay in an 8 bit grayscale. Then, a smaller window at the center is taken and downsized until it reaches the final size of 10x12, same as the real data.

To update the position of these arrays, they are attributed a vertical speed given their horizontal coordinate, and at each frame their new positions are calculated by adding this speed.

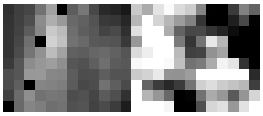

Fig. 1. Real image on the left, synthetic image on the right

The function used to assign these speeds is the hyperbolic tangent $\tanh$, with several settings to tweak its behaviour. To get the labels, we can simply average this speed function over each column of the final window.

Everytime the script is run, it will generate a new folder inside `data/`, where it will put the final frames, the labels, and various other information for debuging and for reproducibility purposes (like a speedplot, a list of settings, and a video of all the frames). At the beginning of the script there is a list of variables used as settings to tweak the behaviour of the generated dataset.

This method gets reasonably close to how the real data looks like. In the precedent description, we have overseen a few simplifications in the behaviour of the synthetic dataset. For instance, the shear layer is always vertical but the real data's shear layer is slightly curved. Another important distinction is that the speed of the structures aren't only along side the $z$-axis in the plasma. Simulating these aspects has been judged too complex, and instead some shortcuts like the negative gaussians are there to try and mitigate this. Overall, the lab seemed satisfied with these synthetic datas.

### C. Data Manipulation and Preprocessing

We have implemented two different data loaders, that are used to organize and load real or synthetic data into the CNN.

Both data loaders organize the frames into 2000-frames datapoints (10x12x2000 values), and shuffle these datapoints using a fixed seed to homogenize the different scenarios. These datapoints are separated into 3 datasets: one for training, one for validation, and one for testing, with default ratios of 64%, 16% and 20% respectively.

The last transformation these datapoints need is to be converted from a 1-channel greyscale to 3-channel, to be compatible with the architectures we used. This was done simply by duplicating the single channel across the two others.

The differences between the two data loaders are where and how they fetch the frames, as they are stored in a different way on disk. Moreover, the dataloader working on real data has to rescale the values of the measures to be 8-bit greyscale images instead of raw brightness measures. This isn't a problem for the synthetic dataloader as the synthetic data was already generated in a convenient way.

Due to time constraints, we only could use part of all the sets for the real data: `t_window`, `r_arr`, `z_arr` and `vz_err_from_wk` were not taken into consideration for our algorithm. Moreover, we had to ignore datasets where the 13th column was missing as described in the real data description, as that would have required us to redo our approach completely.

Finally, we chose to work with inputs of 2000 images, as more images was pushing the limits of our computers in terms of time and memory. The lab estimated that an entire structure would take approximately 18'000 frames to cross the image completely. 2'000 frames should ensure it moves at least one pixel, as the images has a height of 10 pixels.

## III. Model training

### A. The CNN architectures

To perform our regression task, we have decided to work on the architectures defined for video classification in PyTorch, for simplicity and efficiency: we trust the models have been optimized and tested, and it would be easier and safer to implement them rather than develop our own network. The models used are: *ResNet 3D*, *ResNet Mixed Convolution* and *ResNet (2+1)D* [2]. ResNet is a classic neural network that helps solving computer vision tasks [3]. As we have to analyze a sequence of images, this model is appropriate for our problem.

These networks consist of multiple processing layers that perform different operations to process the input, extract and recognise its key features (measurable property or characteristic), and predict the desired outputs. These layers are:

- The convolutional layer, in which features from input images or feature maps are extracted by applying filters, each composed of small kernels.
- The pooling layer, that is similar to the convolutional layer, but performs a specific function, such as taking the maximum or average value in a region, to reduce the complexity and amount of computation of the network
- The fully connected layer, where after all the necessary computation is done, the network regroups all informations from the final feature maps, and generates the output.

The main difference between the three architectures lies in the convolutional layers, where filters are applied distinctively:

- ResNet 3D performs a 3D convolution, the 3D filters are convolved over both time and space dimensions.
- ResNet Mixed Convolution starts with 3D convolution, then at later layers, uses 2D convolution, as at higher level of abstraction, motion or temporal modeling may not be necessary.
- ResNet (2+1)D seperates the 3D convolution into two steps, a 2D convolution in the space dimension followed by a 1D convolution in time, to reduce the computation cost [4].

### B. The training procedure

The main goal we want to achieve when training a model is preparing it sufficiently with randomly generated data in order for it to be ready for the real data. In our case, we start training it on the synthetic data we generated so that we can get a good impression of how well it performs. Only then, we can start training it on the real data, and optimize with both datasets.

The training loop consists of two main phases:

1) **The training phase** In this phase, the model learns how to recognize the key features and where they are, and will update its weights (learnable parameters) with its performance on the dataset. The inputs traverse the whole network, and the model will predict the values for the output, and compare with the labels. Then, the gradients are propagated and the model updates its weights. Note that it only learns with the training set. After the entire set is passed through the network, it then proceeds to the next phase
2) **The validation phase** In this phase, the model simply predicts the output for the validation set, and we compute the loss. This part is to give an estimation how good the model performs at each iteration, to detect any signs of overfitting, and tune hyperparameters later.

Each step is repeated for 30 epochs. The best validation results are used to compare different instances of the model and finetune the hyperparameters. After optimizing and selecting the best performing model, it is then tested on the testing set, which contains data that the model has never been executed with. The testing phase is only to provide us with an evaluation of the final model.

The loss function used to evaluate the model is the MSE (Mean Square Error) loss function, as it is the most commonly used function for regression tasks. As for the optimizer, we decide to choose the optimizer SGD. We tested with Adam, but the model struggled to converge, and resulted in worse losses.

### C. Finding the correct architecture

To find the most suitable model for our task, we decided to train and evaluate with our synthetic dataset, as we only had that available at the time. We see from the last validation results from table I, that ResNet 3D offers the best results. Moreover, while training the models afterwards, our computers were unable to handle the ResNet MixedConv and ResNet 2+1D, as the memory load is too heavy with some set of hyperparameters, and would restrain us for optimization. We therefore decide to use and optimize ResNet 3D.

| Architecture | Validation loss |
|---|---|
| ResNet 3D | 1.021 |
| ResNet MixedConv | 3.792 |
| ResNet 2+1D | 3.284 |

TABLE I
LAST VALIDATION RESULTS FOR THE ARCHITECTURES AFTER 30 EPOCHS.

To finetune our model, we decided to start by training our model on the synthetic dataset, to have a first impression on how the model will behave and which hyperparameters to optimize. The hyperparameters we chose to finetune are :

- The learning rate *lr*, which describes how fast the model adapts to the problem, how much should it update its weights based on the estimated error. Choosing a learning rate that is too high can cause the model to converge too quickly to a sub-optimal solution or overfit the training data, while a too low learning rate results in a long training process, or can even cause the model to be stuck.
- *batch_size*, which defines the number of samples that will be trained at the same time, and affects how the gradient is calculated when correcting the weights. In short, bigger batches means the gradient should fluctuate less and be more accurate, at the expense of memory required. But it doesn't necessarily means a better model, as a noisier gradient might lead to a more robust model.

| | 0.005 | 0.01 | 0.02 | 0.03 | 0.04 | 0.05 | 0.06 |
|---|---|---|---|---|---|---|---|
| 2 | 3.77 | 2.81 | 1.50 | 2.54 | | | |
| 4 | S | M | | J | | | |
| 8 | (S) | | | | | | |
| 16 | | | 5.76 | 4.02 | 3.97 | 3.12 | 4.00 |

TABLE II
BATCH SIZE (COLUMN) VS LEARNING RATE (ROWS) ON SYNTHETIC DATA

| | 0.005 | 0.01 | 0.02 | 0.03 | 0.04 |
|---|---|---|---|---|---|
| 2 | 21.20 | 18.52 | 19.21 | 20.69 | 27.09 |
| 4 | 20.45 | 19.64 | 21.29 | 21.50 | 21.54 |
| 8 | | 18.91 | 18.90 | 18.21 | 19.80 |
| 16 | | 19.89 | 18.50 | 19.35 | |

TABLE III
BATCH SIZE (COLUMN) VS LEARNING RATE (ROWS) ON THE REAL DATA

## IV. RESULTS

## V. CONCLUSION

## REFERENCES


[1] M. J. Park and M. Sacchi, "Automatic velocity analysis using convolutional neural network and transfer learning," 2019. [Online]. Available: https://www.researchgate.net/publication/336339562_Automatic_velocity_analysis_using_convolutional_neural_network_and_transfer_learning

[2] "Models definition and source code." [Online]. Available: https://pytorch.org/docs/stable/torchvision/models.html#video-classification

[3] K. Hara, H. Kataoka, and Y. Satoh, "Can spatiotemporal 3d cnns retrace the history of 2d cnns and imagenet?" *arXiv preprint*, vol. arXiv:1711.09577, 2017.

[4] D. Tran, H. Wang, L. Torresani, J. Ray, Y. LeCun, and M. Paluri, "A closer look at spatiotemporal convolutions for action recognition," *arXiv preprint*, vol. arXiv:1711.11248, 2018.