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 optimise 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 the shear layer. GPI analysis is a 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 velocity of the moving, turbulent structures captured by GPI. We quickly decided to go for a convolutional neural net working on sequences of frames, as it seems a good approach for complex tasks and apt for 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 (12x10xl 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 pixels 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 column, which means this column will have flows going both way. 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 generate this in the following way: we draw a large canvas (by default 480x480), and spawn in randomly 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 8 bit greyscale. 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

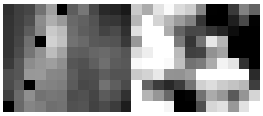

Fig. 1. Real image on the left, synthetic image on the right

frame their new positions are calculated by adding this speed. The function used to assign these speeds is the hyperbolic tangent, with several settings to tweak its behavior. 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 behavior of the generated dataset.

This method gets reasonably close to how the real data looks like; in the precedent description, you might have spotted a few simplifications in the behavior of the synthetic dataset: for instance, the shear layer is always vertical; the real data's shear layer is slightly curved. Another importante distinction is that the speed of the structures aren't only in Z 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 satified with these synthetic datas.

C. Data Manipulation and Preprocessing

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

Both dataloaders organize the frames into 2000-frames datapoints (10x12x2000 values), and shuffles 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 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 other.

The differences between the two dataloaders 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 datas were generated in a convinient way already.

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 choose 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 18'000 frames to cross the image completely; 2'000 frames should ensure it moves at least one pixel, as the images is 10 pixels in height.

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 optimised 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*. ResNet is a classic neural network that helps solving computer vision tasks [2]. 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:

- Convolutional layer, in which features from input images or feature maps are extracted by applying filters, composed of small kernels.
- Pooling layer, that are similar to convolutional layer, but perform 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
- 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 cost of computation. [3]

B. The training procedure

The training loop consists of two main phases:

a) 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. This part essentially consists of two main phases: a forward phase and a backward phase.

- In the forward phase, the input traverses the whole network. The model will predict the values for the output, and compare with the labels.
- In the backward phase, the gradients are propagated throughout the network (back propagation algorithm) and update the weights of the model.

The model learns only with the training set. After the entire set is passed through the network, it then proceeds to the next phase

b) 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.

These steps are repeated for 30 epochs. The best validation results are used to compare different instances of the model and finetune the hyperparameters. After optimising and selecting the best performing model, it is then tested on the model is 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.

C. Finding the correct architecture

To find the most suitable model for our task, we decide 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 2+1D offers the best results. However, while training the models afterwards, our computers are unable to handle the ResNet MixedConv and ResNet 2+1D, as they are too memory heavy with some set of hypterparameters, for example even a batch size of 8 on the synthetic dataset is too much for us, probably linked to how the networks preload the inputs, which means the memory load would be even heavier with the real dataset. So we decide to use and optimise ResNet 3D instead.

Architecture	Validation loss
ResNet 3D	6.687
ResNet MixedConv	6.347
ResNet 2+1D	4.594

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

D. Training the model on synthetic data

To finetune our model, we decide 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 optimise. The hyperparameters we choose to finetune are :

- *lr*, the learning rate, which is 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 even cause the model to be stuck.
- *batch_size*, which defines the number of samples that will be trained at the same time, and it 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 necesserily mean a better model, as a noisier gradient might lead to a more robust model.
- *gamma*, which controls how fast the learning rate decreases over the epochs, so that the model can converge to a solution.

E. Training the model on the real data

(This section is subject to change) From the dataset and analysis provided to us by the lab, we know that for a blob to traverse from the bottom to the top of the canvas, it takes around [...] ms, or 17800 frames. That means we could give the model inputs of size 17800 images, and the model can follow the blob throughout the sequence and predict its velocity. However, due to hardware limitation, as we are training the models directly on our laptops, our GPUs are unable to handle such huge inputs. So we settle with 2000 frames per input.

After receiving the real dataset and understanding how it works,

IV. RESULTS

	0.005	0.01	0.02	0.03	0.04
2	21.20		28.58	23.70	20.13
4			21.29	21.50	20.87
8			18.90	18.21	19.80
16		19.89	18.50	19.35	

TABLE II
BATCH SIZE (COLUMN) VS LEARNING RATE (ROWS)

A. Parameters learnt with synthetic data

We describe in this part the tests we have done in order to find the best parameters to use for the real data. Parameters we have tried to find are typically the γ and the learning rate. We have based ourselves on the training and validation losses.

We have for the purpose, created a python script that goes through all different parameters to test and generates a plot according to the matter. With a γ varying from 0.05 to 0.5 and a learning rate varying from 0.005 to 0.1, we get plots that ressemble each other but generally have two main different lookings.

1) The first one has a very steep exponential decrease for the first 10 to 15 epochs and then decreases quite slowly but has an average loss that is quite small.
2) The other, starts with a smaller training loss and a nearly zero validation loss. However, the validations loss pattern is quite different: it increases for a few epochs and then decreases almost linearly but is more significant than the preceding one after the 30 epochs. It can also start very high and diminish extremly fast during the first few epochs, before increasing a little again.

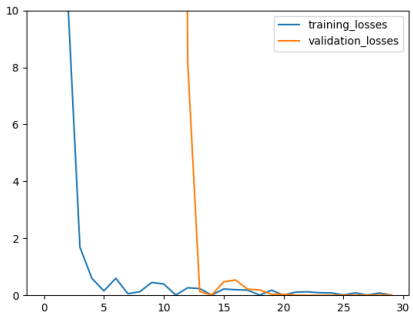

Fig. 2. Best parameters for first kind of plot

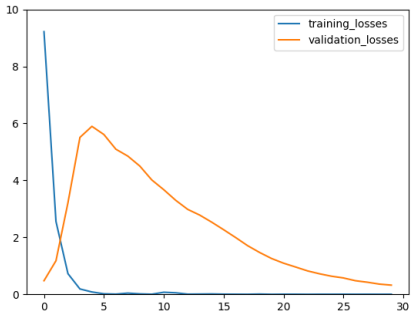

Fig. 3. Best parameters for second kind of plot

Note that the first plot above is obtained with a learning rate of 0.005 and $\gamma = 0.05$ whereas the second one has a learning rate of 0.05 and $\gamma = 0.5$.

B. Interpretation of the results

The most important factor we have to take into account is the losses identified after a bit of training has happened, usually, after 20 epochs is a good time estimator of how our algorithm is performing. In the two above plots, we see that in one case we have an enormous loss margin at the start of the algorithm but it gets relatively small quite quickly whereas in the other case, the validation loss doesn't have time to converge enough to provide us sufficient results. Therefore, in that case keeping $\gamma = 0.05$ and a learning rate of 0.005 is the best solution.

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] 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.

[3] 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.