Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Aerospaziale

Tesi di Laurea Magistrale

Application of Convolutional Neural Networks to Particle Image Velocimetry

Relatori:

Prof.ssa Sandra Pieraccini

Dott. Gioacchino Cafiero

Prof. Gaetano Iuso

Candidata:

Luisa Di Monaco

Aprile 2020

Abstract

In the last decade, deep learning architectures have overwhelmingly outperformed the state-of-the-art in many traditional computer vision tasks as image classification and object detection, so that it becomes interesting to test them on new tasks such as regression. Particle image velocimetry (PIV) is an experimental fluid dynamics technique that involves image analysis. PIV processing represents a complex regression problem. Therefore, it is interesting to test the ability of artificial neural networks (ANNs) to perform PIV. The first feasibility study on ANN application to standard PIV was "Performing particle image velocimetry using artificial neural networks: a proof-of-concept" published in 2017 by Rabault et al. The present work aims to build and train a convolutional neural network (CNN) to perform standard PIV and to test the model on synthetic and real PIV data. More dense particle images and higher displacement than those studied by Rabault et al. are considered. Both training and test phase are performed using Matlab 2019b with Deep Learning Toolbox. Performances are promising on synthetic data, but still unsatisfactory on real data. Accuracy can be improved by increasing image resolution and by computing the mean field from both several particle images and predictions of models trained differently, despite higher computational cost.

A Laura,
che quando si parla di algoritmi e coefficienti
sa sempre che cosa dire.

Contents

List of tables

List of figures

Abbreviations

ANN	Artificial Neural Network
CLAHE	Contrast-Limited Adaptive Histogram Equalization
CNN	Convolutional Neural Network
HPC	High Performance Computer
HWA	Hot Wire Anemometry
LDV	Laser Doppler Velocimetry
PIV	Particle Image Velocimetry
RGB	Red Green Blue
RMSE	Root Mean Square Error
RMSProp	Root Mean Square Propagation
SGDM	Stochastic Gradient Descent with Momentum

1. Introduction

This section provides an overview on major achievements in deep learning, particularly focusing on advances in convolutional neural networks (CNNs) in section 1.1 and recent applications of CNNs to particle image velocimetry (PIV) in section 1.2. Objectives of the present work are then explained (section 1.3).

1.1. Background

The first simple artificial neural network was developed in the 1950s and 1960s by F. Rosenblatt [1], inspired by the earlier work by W. McCulloch and W. Pitts [2]. It was called perceptron. Perceptron is a linear binary classifier (section 2.2.2). Multi-layer perceptrons are called neural networks. Neural networks were thought to be fundamentally limited until 1986 when backpropagation was introduced. The backpropagation algorithm (section 2.2.3) was originally proposed in the 1970s, but its importance wasn't fully appreciated until a 1986 paper by D. Rumelhart et al. [3]. That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural networks to solve problems which had previously been insoluble. However, a limited number of problems could be solved using neural networks until 2006. Since 2006, a set of techniques has been developed that enables learning in deep neural networks by improving algorithms for backpropagation [4]. They have allowed much deeper and larger networks to be trained so that people now routinely train networks with 5 to 10 hidden layers [5]. It turns out that they perform far better on many problems than shallow neural networks with just a single hidden layer [5]. Today, the backpropagation algorithm is the workhorse of learning in neural networks [5]. In 2010-2012 the so-called deep learning revolution began: deep learning models started to make substantial progress on several problems. The deep learning revolution came to show that neural networks were not insufficiently powerful, as most believed until then, just lacking the required amount of computing power and training data to perform well.

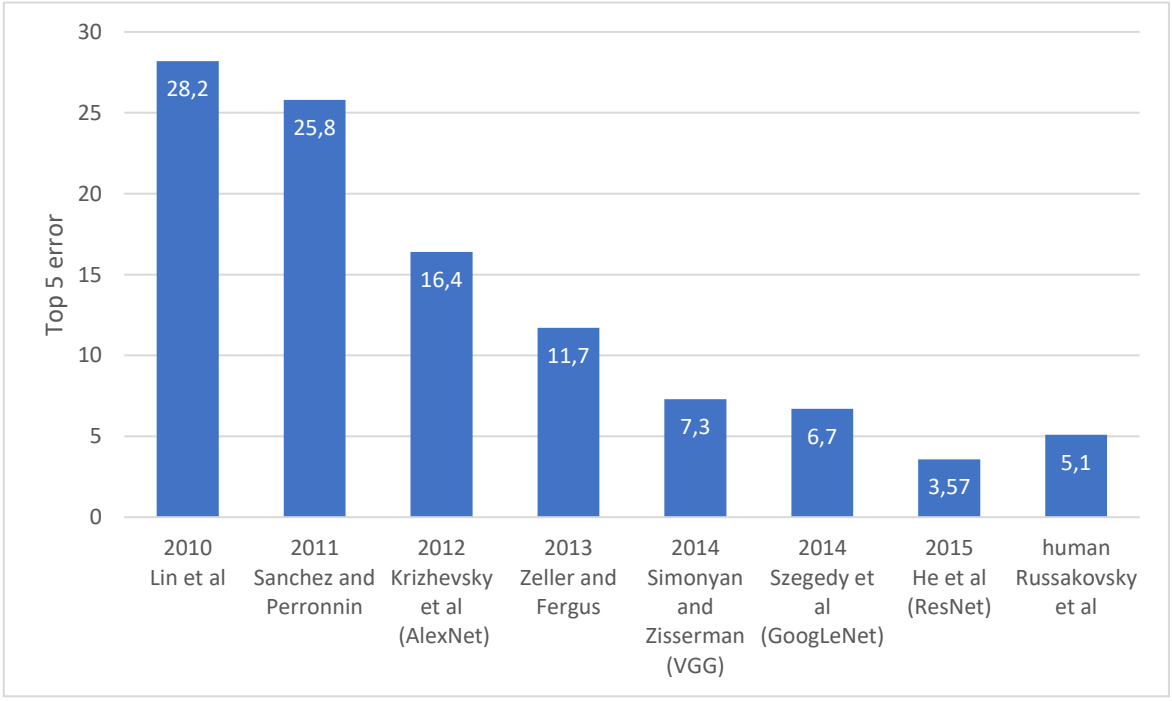

Figure 1: CNN progress in image classification [6]

CNNs had a big role in this revolution. In 2012, a CNN model by A. Krizhevsky et al. won the ImageNet Image Classification Challenge (1000 object classes, more than 1 000 000 images to be classified) by a large margin [7]. ImageNet is a benchmark in object category classification and detection. In the following years, CNN performances were rapidly improved by implementing deeper models (figure 1). In 2015 CNNs exceeded human performances [6].

Since for the last decade deep learning architectures have overwhelmingly outperformed the state-of-the-art in many traditional computer vision tasks as image classification or object detection, it becomes interesting to test them on new tasks such as regression. Recently, CNNs have been tested on more complex regression problems [8, 9, 10]. First results reinforce the hypothesis according to which a general-purpose network architecture adequately tuned can yield results close to the state-of-the-art without having to resort to more complex and ad-hoc regression models [11].

PIV is an experimental fluid dynamics technique that involves image analysis [12]. It deals with the reconstruction of velocity fields from image pairs of particles that are seeded in fluid flows. At the state of art, PIV processing is usually performed via the cross-correlation method [13, 12, 14]. It represents a complex regression problem.

1.2. Related works

With the development of deep learning techniques, the introduction of big data and the improvement of computational performances, designing a deep neural network for velocity field reconstruction become a promising direction.

A proof-of-concept on this topic is provided by Rabault et al. [15]. The authors designed both a fully connected neural network and a convolutional neural network to perform end-to-end PIV for the first time and compared the network predictions with state-of-the-art PIV software. The error between their predictions and the velocity values used for generating the particle images is higher than state-of-the-art PIV codes, but much better than what would be expected from single pass PIV. This is a good result considering that current state-of-the-art PIV software have been developed for decades. Moreover, the author noticed that neural network models are better at using efficiently GPUs than commercial PIV software. The philosophy of the network models is similar to the cross-correlation method with interrogation windows of size 32x32 pixels.

A similar approach is used by Y. Lee et al. with interrogation windows of size 64x64 pixels and a four-level regression deep CNNs [16]. At each level, the network is trained to predict a vector from two particle image patches. The low-level network is skilled at large displacement estimation and the high-level networks are devoted to improving the accuracy. Achieved performances are competitive with state-of-the-art PIV software. However, the proposed model doesn't reduce the execution time with reference to standard PIV software.

Beside cross-correlation, variational optical flow methods have been developed to extract velocity fields from particle images [17]. While they provide more dense velocity field than the cross-correlation approach, they are more time consuming and sensitive to noise [18]. Nevertheless, optical flow methods inspired the computer vision community to further develop some established deep learning techniques for optical flow estimation suitable for rigid motion [19, 20, 21] to PIV applications [22, 23]. Compared to the cross-correlation method, the network by S. Cai et al. [23] is much better at resolving small scales and provides much dense velocity reconstructions. Compared to variational methods, it results more efficient. It is also

robust against noise. However, it doesn't outperform state-of-the-art PIV methods at the moment. According to the authors, more training is needed to make the model more robust for different flow motions and for real PIV images. Moreover, the deep neural network requires large computing memory on GPU and memory overflow can be encountered especially while processing high resolution images. Therefore, it may be a trouble to replicate the model on ordinary computers.

1.3. Objectives

The present work aims to build, train and test a CNN to perform end-to-end PIV (figure 2). Synthetic data are used for the training phase. The trained network is tested on synthetic and real PIV data. Standard PIV and stationary 2-dimensional velocity fields are considered. The paper by Rabault et al. is taken as main reference to build and train the network [15]. Effects of image resolution, computing the mean field on both different particle image pairs and different networks are evaluated. Performances on both a larger range of pixel displacements and higher particle image density than the paper by Rabault et al. [15] are investigated.

A secondary aim is to test Matlab 2019b with Deep Learning Toolbox as deep learning framework.

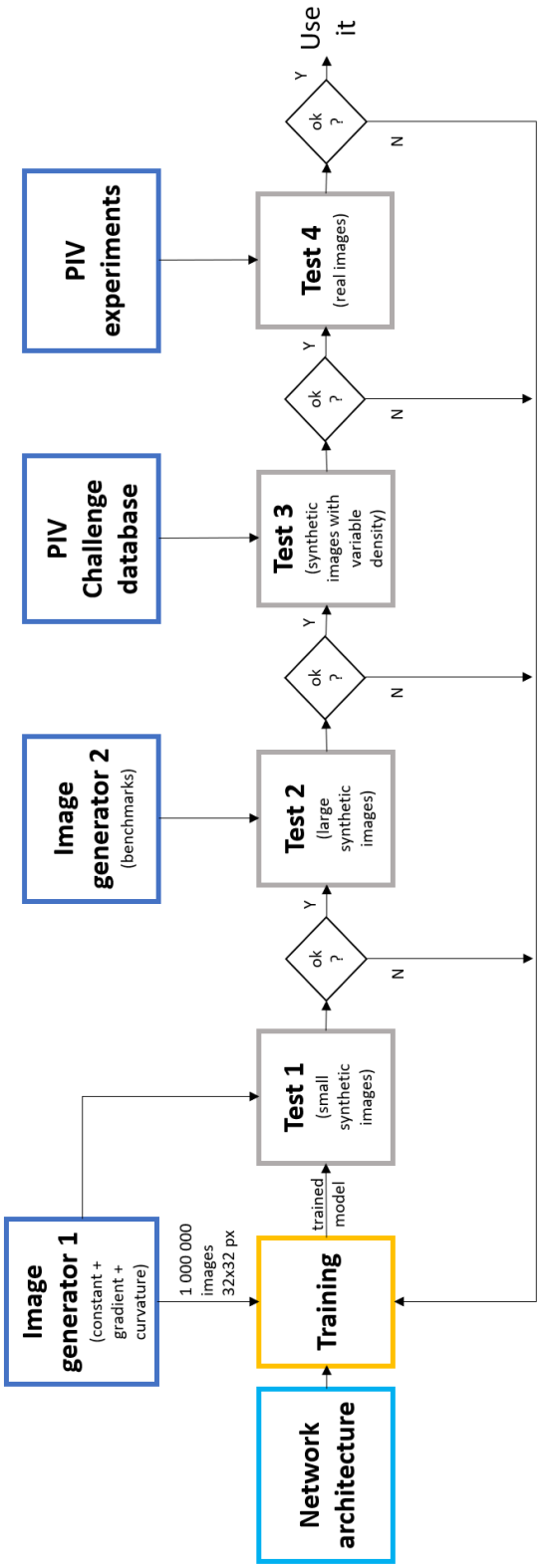

Figure 2: concept

2. An overview on the mathematical model: neural networks

CNNs are a class of deep neural networks working particularly well for image analysis. It is common to refer to the field of deep neural networks as deep learning. Deep learning methods are part of a broader family of machine learning methods that perform specific tasks without using specific instructions. A machine learning algorithm is in fact an algorithm that is able to learn from data. In this section, the concepts under machine learning are introduced focusing on potentials and challenges (section 2.1). The structure of deep learning algorithms and training strategies are briefly presented (sections 2.2). Section 2.3 focuses on CNNs.

2.1. Learning algorithms

2.1.1. What is machine learning?

According to Goodfellow et al., machine learning is essentially a form of applied statistics with increased emphasis on the use of computers to statistically estimate complicated functions and a decreased emphasis on proving confidence intervals around these functions [24].

The aim of machine learning is to complete the learning process. According to the following definition by Mitchell T. M. : "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" [25]. In this definition task, performance and experience are intended as described below.

Task

Learning is the process of attaining the ability to perform a task. Machine learning tasks are usually described in terms of how the machine learning system should process an example. An example is a collection of features that have been quantitatively measured from some object or event that we want the machine learning system to process [24]. It is represented by a vector $\boldsymbol{x} \in \mathbb{R}^n$ where each entry x_i is a feature. Typically, machine learning tasks are difficult to be solved using traditional programming. Some of the most common tasks are classification and regression. In classification tasks, a list of categories is given to the machine learning algorithm and it is asked to identify which category the input belongs to. In regression tasks, the machine learning algorithm is asked to predict some numerical value given an input.

Performance

A quantitative measure of the algorithm performance is needed to evaluate its ability to perform a specific task. The choice of performance measure depends on the task and the specific application. For classification, accuracy (the proportion of examples for which the model produces the correct output) or error-rate (the proportion of example for which the model produces the incorrect output) can be useful. For regression, mean absolute error or mean relative error with standard deviation values can be effectively used [24].

Experience

Machine learning algorithms can be broadly categorized as unsupervised or supervised by what kind of experience they are allowed to have during the learning process. A dataset is a collection of many examples that is experienced by the algorithm during training. Data are usually stored in matrices. In unsupervised learning, the algorithm experiences a dataset containing many features and is expected to learn the probability distribution or some interesting properties of that distribution of the structure of the dataset. Supervised learning algorithms experience a dataset of examples containing features. Each example is associated with a label or numeric values. The target of supervised neural networks is learning how to map a given input example to the correct label or numeric values. Traditional examples are classification or

regression for supervised learning and clustering for unsupervised learning. However, the lines between supervised and unsupervised learning are in facts blurred [24].

2.1.2. Challenges in machine learning

The central challenge in machine learning is to build an algorithm able to generalize, i.e. that performs well on unobserved inputs and not just on the training dataset [24]. This ability is merely called generalization.

As in optimization problems, in machine learning we want to reduce the error and improve performances on the training set. However, this is not enough: we want that the algorithm performs well on new input as well. The test error is defined as the expected value on a new input and it is typically measured on a test dataset of examples collected separately from the training set. Machine learning algorithms aim to make the test error small.

The accuracy of a neural network depends on how much the training process:

- make the training error small;
- make the gap between training and test error small.

Underfitting occurs when the training error is too high. Overfitting occurs when the gap between training error and test error is too large. Both underfitting and overfitting are due to bad generalization. In the first case, it relates to poor learning. In the second case, it relates to excessively specific learning on training data.

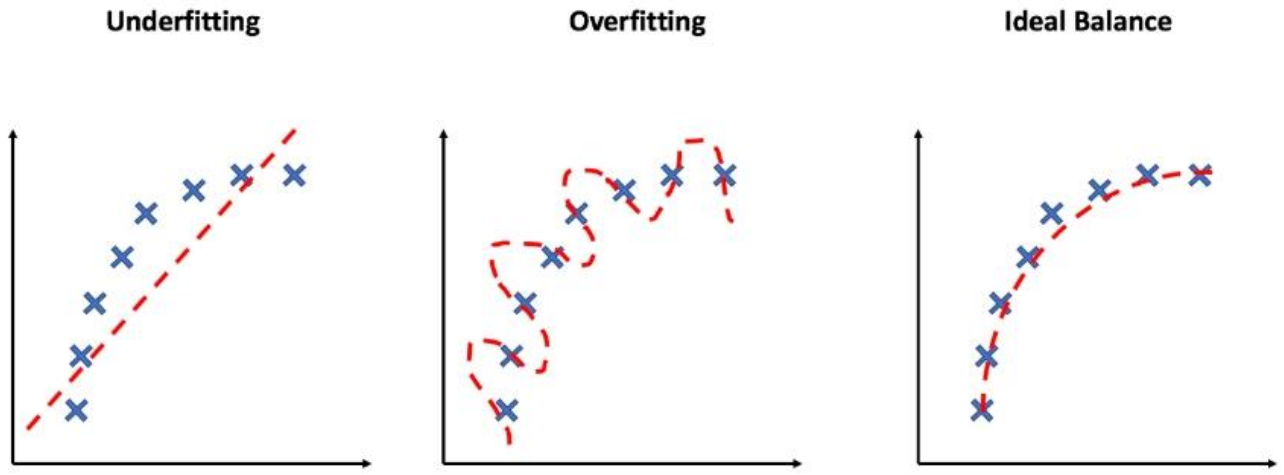

Figure 3: visual demonstration of underfitting, overfitting and balance (figure taken from [26])

We can control whether a model is more likely to overfit or underfit by altering its capacity that is the ability to fit a wide variety of functions [24]. Models with too low capacity struggle to fit the training set. Models with too high capacity tend to overfit instead. This happens because they memorize properties of the training set that are not so useful to perform well on the test set. So, capacity has to be appropriate to the task of the algorithm. An insufficient capacity makes the model unable to solve complex tasks, on the other hand high capacity models can solve complex tasks, but they cause the model to overfit when the assigned task can be solved using a less capable model.

It is possible to change the representational capacity of the model by changing the set of functions that the learning algorithm is allowed to select as solution, i.e. by changing the functions in the hypothesis space. The hypothesis space is the space of possible hypotheses. Each hypothesis is a candidate model that

approximates a target function for mapping inputs to outputs. Including more complex functions in the hypothesis space means increasing the number of input features and the model capacity.

The effective capacity is often different by the representational one because of the optimization algorithm. Actually, that algorithm doesn't select the best model of the assigned set of functions, but only one that significantly decreases the training error.

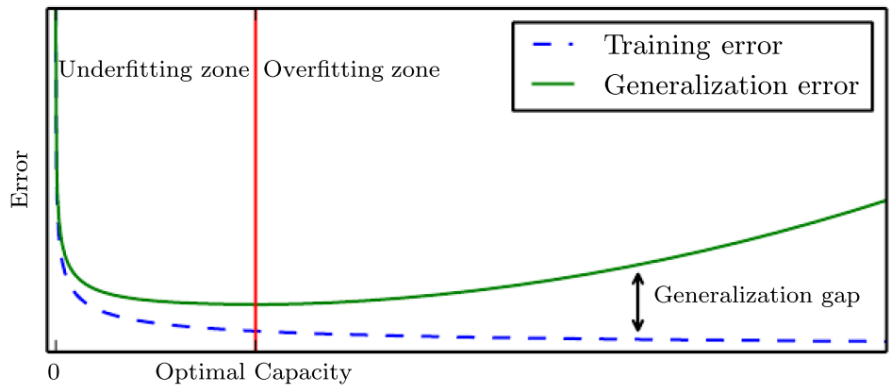

Figure 4: effect of capacity on training and test errors (figure taken from [24])

Statistical learning theory shows that the discrepancy between training error and generalization error is bounded from above by a quantity that grows as the model capacity grows but shrinks as the number of training examples increases [27]. Typically, the test error has a U-shaped curve as a function of the model capacity. It is possible for the model to have optimal capacity and yet still have a large gap between training and generalization error. In this situation, it is often possible to reduce this gap by gathering more training examples [24].

The goal of machine learning research is not to seek a universal learning algorithm or the absolute best learning algorithm. Instead, it tries to understand which kind of algorithm perform well on the specific task we are trying to solve [24].

Methods for modifying a learning algorithm are:

- changing the kind of functions in the hypothesis space;
- changing the amount of functions in the hypothesis space;
- regularization, i.e. giving a preference for one solution in the hypothesis space to another.

Regularization can be implemented using different approaches:

- put extra constrains on the machine learning model;
- add extra term to the cost function;
- use an ensemble method combining multiple hypothesis to explain the same training data.

Using regularization, the algorithm can select either preferred or not preferred solution, but the second one is chosen only if it fits the data much better than the first one. Regularization aims to reduce the test error, but not the training error. Essentially, it is a way to reduce the gap between the two kinds of error.

Three situations may arise during training [24]:

- the model class excluded the true data generating process: the model underfits and induces bias;

- the model class matched the true data generating process: the model is appropriate;
- the model class matched the true data generating process, but also many others generating processes: the model overfits resulting in low bias and high variance.

Regularization tends to reduce variance and increase bias. Effective applications of regularization reduce variance significantly while not overly increasing bias. This leads to move from the third to the second training situation. A common example of regularization is the "weight decay" or “L2 regularization” that bounds the maximum absolute values of the learning parameters of the algorithm (also called weights), setting a preference for a particular subset of functions in the hypothesis space.

Although the learning ability of machine learning algorithms depends mainly on weights, most of the algorithms are characterized also by fixed parameters. They are called hyperparameters and are very problem dependent. They can be used to control the behavior of the algorithm, for example to prevent underfitting and overfitting. It is necessary to try their values all out and see what works best [28]. Realizing what “work best” means in every specific problem can be misleading. Li et al. show pro and cons of different choices [28]:

- It may seem logical to choose hyperparameters that work best on the considered dataset. However, this choice typically works perfectly on training data and bad on new data.

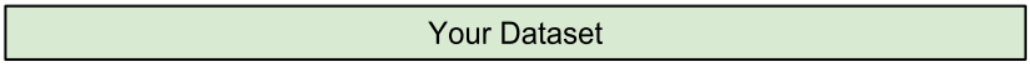

Figure 5: choice of hyperparameters, dataset (figure taken from [28])

- The second idea is to split data into training set and test set. Hyperparameters that work best on test data are chosen. This choice is typically better than the first one but give us no idea about performances on new data.

Figure 6: choice of hyperparameters, training and test sets (figure taken from [28])

- The third idea is to split data into training, validation and test sets. Hyperparameters that work best on validation data are chosen, then performances are evaluated on the test set. This is the most common approach to train deep learning algorithms.

Figure 7: choice of hyperparameters, training, validation and test sets (figure taken from [28])

- The fourth idea is called cross-validation. It consists in splitting data into several groups and repeating the training phase using each group as validation set. The best value of each hyperparameter is set averaging results of all trainings. This is a useful method for small datasets, but it is not common in deep learning because of its high computational cost.

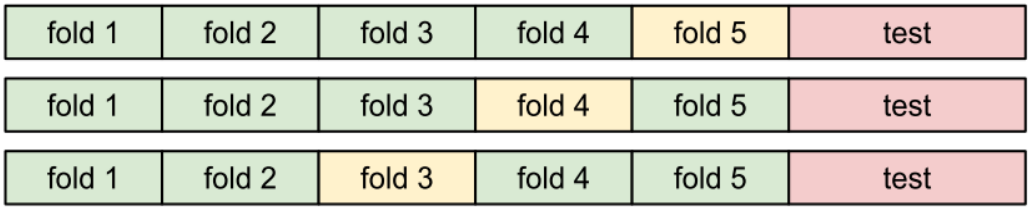

Figure 8: choice of hyperparameters, cross-validation (figure taken from [28])

To recap, in deep learning typically we start with a training dataset and we have to make a prediction on the test set. We choose hyperparameters using a validation set and applying a trial and error procedure.

Usually, the test set is only used once at the very end of training [24, 28].

2.1.3. Building a machine learning algorithm

Nearly all deep learning algorithms can be built using a combination of the following elements:

- a dataset containing inputs and targets;
- a model that makes predictions starting from inputs;
- a cost function that quantifies the dissimilarity between predictions by the algorithm and targets;
- an optimization procedure that allows to modify the training parameters of the model in order to minimize the cost function.

Any of these elements can be chosen most independently from the others obtaining a variety of algorithms. According to Goodfellow et al., it is important to recognize that most machine learning algorithms can be described in this way [24]. This view helps to see the different algorithms as part of a taxonomy of methods for doing related tasks that work for similar reasons, rather than as a long list of algorithms that each have separate justifications [24].

If the model is nonlinear, then most loss functions cannot be optimized in closed form and an iterative procedure must be implemented. In deep neural networks, the optimization procedure combines a backpropagation algorithm and an optimization algorithm. Backpropagation is essentially an efficient way to compute the derivatives basing on the chain rule (section 2.2.3). It is used to compute the gradient of the cost function with respect to each parameter of the network [4, 3]. After that, an optimization algorithm, such as stochastic gradient descent or Adam, updates weights basing on the computed gradients [29, 30].

2.2. Deep neural networks

2.2.1. Fully connected neural networks

Neural networks are algorithms aiming to approximate some function f^* by composing together many different functions and taking advantage of statistical generalization.

A neural network model is associated with a graph describing how the functions are composed together. In most cases, different functions are connected in a chain structure where each function is called layer. For example, a neural network f with four layers (figure 9) shows the following structure:

$$f(\boldsymbol{x}) = f_3\left(f_2\left(f_1(\boldsymbol{x})\right)\right)$$

where $\boldsymbol{x}$ indicates the inputs and f_i indicates the layer i.

For each function, the dimension of the codomain is equal to the number of units in the layer the function relates to. For example, the functions that make up the network in figure 9 are characterized by the following domains and codomains:

$$f_1: \mathbb{R}^3 \to \mathbb{R}^4$$
$$f_2: \mathbb{R}^4 \to \mathbb{R}^4$$
$$f_3: \mathbb{R}^4 \to \mathbb{R}.$$

Each intermediate layer is called hidden layer and is typically a vector consisting of many units that act in parallel. Each unit represents a vector-to-scalar function: it receives inputs from many other units located in the previous layer and computes its activation value. All the units of the previous layer are connected to each unit of a layer in the case of fully connected neural network.

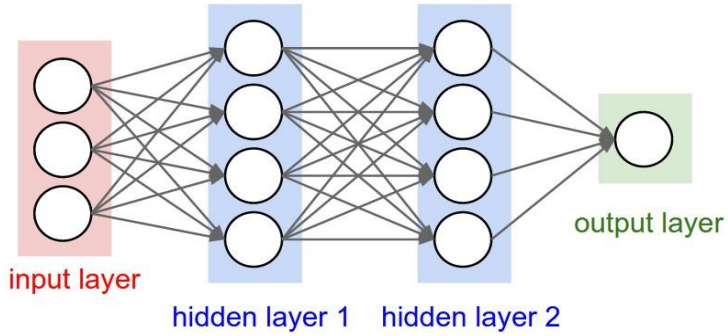

Figure 9: 4-layers fully connected network (figure taken from [28])

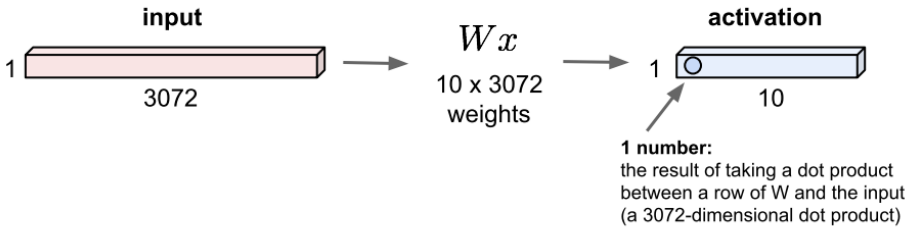

Figure 10: fully connected layer (figure taken from [28])

The overall length of the chain and the number of units per layer determine the depth and the width of the model respectively. The combination of the arrangement of neurons into layers and the connection patterns between layers and activation functions is called network architecture. The network architecture determines how the network transforms its inputs into the outputs. In feedforward neural networks, the information flows from inputs through hidden layers and finally to outputs without feedback connections. On the contrary, feedbacks are used in recurrent neural networks. The present work is focused on a kind of feedforward networks: CNNs.

The nonlinearity of the model is achieved using activation functions. Typically, each layer f_i includes a linear transformation controlled by learnable parameters (weight and biases) and a fixed nonlinear transformation σ called activation function:

$$f_i(\boldsymbol{x_{i-1}}) = \sigma(W_i \boldsymbol{x_i} + \boldsymbol{b_i})$$

where:

- $\boldsymbol{x_{i-1}} \in \mathbb{R}^n$ is the input vector of the i -th layer and the output vector of the $(i-1)$th layer;
- $W_i \in \mathbb{R}^{n \times m}$ is the weight matrix of layer i, where m is the number of units of the layer;
- $\boldsymbol{b_i} \in \mathbb{R}^m$ is the bias vector of the layer i;
- σ, with an abuse of notation, is the element-wise application of a scalar real function $\sigma: \mathbb{R} \to \mathbb{R}$.

Activation functions can be chosen from a large set of functions. The most popular choices are shown in figure 11. At the moment, the default recommendation for use with most feedforward networks is the rectified linear unit (ReLU) [31, 28]. Using ReLU, each function of the graph is a piecewise linear function. Therefore, ReLU activation functions are able to model nonlinear behavior, but at the same time are not so far from linearity and this is why they allow the networks to preserve some properties of linear functions that make the training process easier [24].

Before ReLU introduction, sigmoid activation function was widely used [5]. However, it is affected by saturation, so that it is sensitive to the input variation only in a limited range centered at zero. This property is very undesired for an activation function [24, 28].

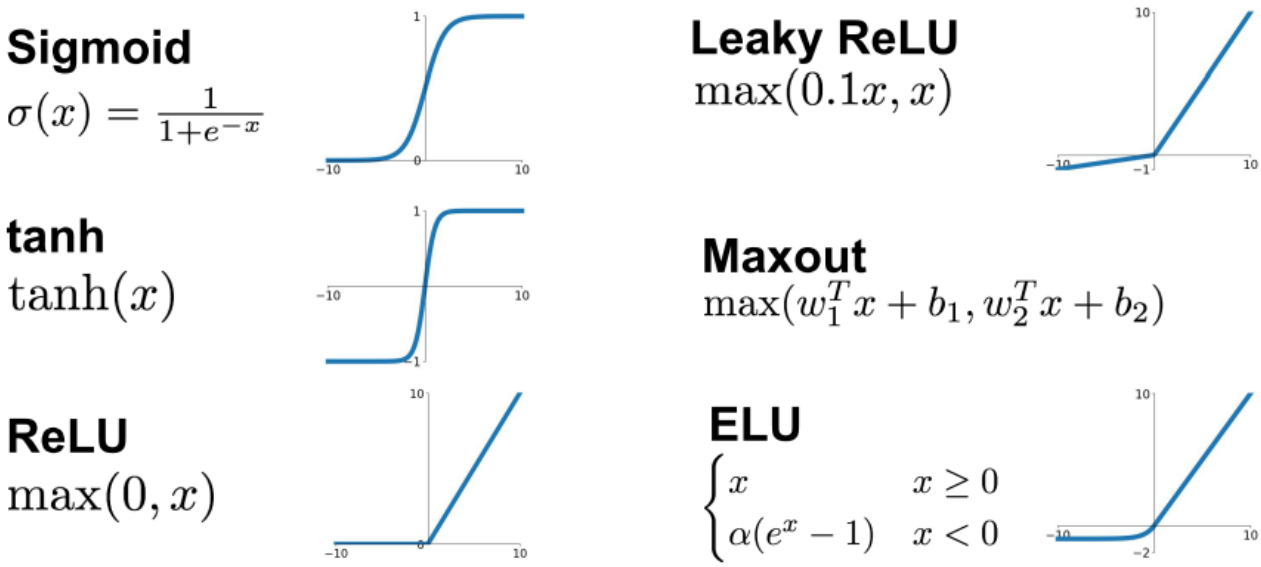

Figure 11: most popular activation functions (figure taken from [28])

Using ReLU, only positive values would pass through the activation function while all negative values are set to zero. ReLU outperforms sigmoid and tanh activation functions with much faster training speed due to non-saturation at 1.

Parametric Rectified Linear Unit (PReLU) is proposed to generalize the traditional rectified unit (ReLU) [32]. It is convenient for cases where there should be a parametric penalty for negative values. The introduced penalty term is:

$$\max(ax, x),$$

where a is the parameter.

PReLU works as ReLU when $a = 0$ and as Leaky ReLU when $a = 0.01$**.** However, PReLU and ReLU always differ because the PReLU parametric value can be learnt during the training phase.

2.2.2. Strengths and weaknesses of neural networks

Neural networks, such as all machine learning algorithms, need a training phase to become operational.

During training, parameters in the model are progressively modified in order to drive the prediction $f(x)$ to match the target $f^*(x)$. The training examples specify directly what the output layer must return at each training inputs, but don't give any information about the behavior of each single hidden layer. For this reason, an optimization procedure is needed. Due to the nonlinearity of the model, training neural networks requires an iterative optimization algorithm. Linear machine learning models can fit functions efficiently and reliably, in closed form or with convex optimization [24]. However, their capacity is limited to linear functions. Neural networks are highly flexible and can successfully approximate more functions paying the price of a more complex and more computationally expensive training phase. In effect, neural networks were demonstrated to be universal approximators [33]. This really means that for every function exists a neural network able to approximate it with an arbitrary degree of accuracy. However, nothing can be a priori known about how wide and deep the network should be and, similarly, we can't know if a training algorithm will be able to effectively train the model. Furthermore, a feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and can fail to generalize to new inputs. Using the same number of parameters, deeper networks seem to learn and generalize better than shallow models. This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together [34]. An efficient network architecture must be found via trial and error using a validation set for training monitoring.

Neural network advantages can be seen comparing neural networks to linear classifiers on complex classification tasks. A linear classifier is a simple machine learning algorithm to approximate functions using a set of weights W and bias $\boldsymbol{b}$ in the form:

$$f(\boldsymbol{x}) = W\boldsymbol{x} + \boldsymbol{b}.$$

This kind of algorithm is able to distinguish points that can be separated by a line in the plane (input, target). A neural network has a more flexible structure that allows the algorithm to apply some feature transformations in order to make points separable by a line (figure 12). This is why neural networks are more powerful than other simple machine learning algorithms.

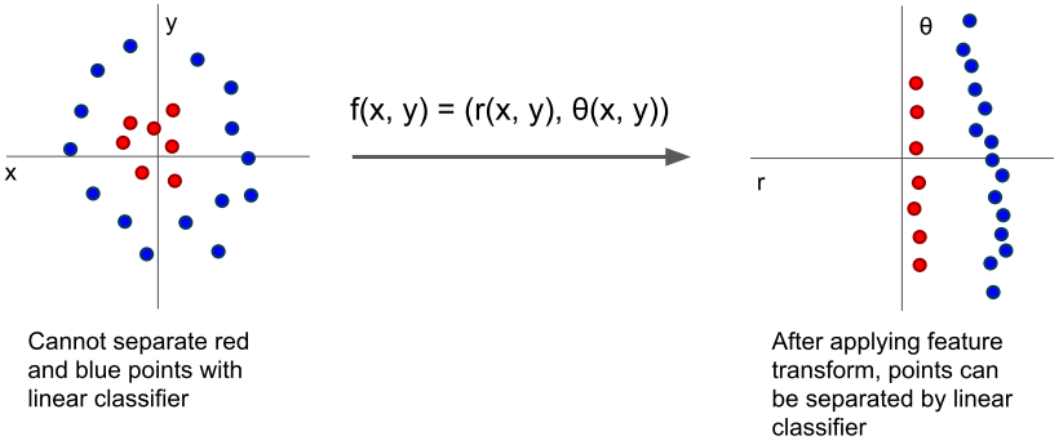

Figure 12: abilities of neural networks (figure taken from [28])

2.2.3. Training of neural networks

According to Andrej Karpathy, it is allegedly easy to get started with training neural networks thanks to numerous libraries and frameworks that can give the false impression that deep learning is plug and play. A lot of people have personally encountered the large gap between “here is how a convolutional layer works” and “our convnet achieves state of the art results”. If you insist on using the technology without understanding how it works you are likely to fail [35].

Neural network training is an iterative process based on the computation of a loss function and its minimization adjusting the learning parameters. Backpropagation is used to compute the gradient of the loss function with respect to weights and biases of the network. Then, the optimization algorithm updates parameters (weights and biases) basing on the loss gradient (returned by the backpropagation) and a particular hyperparameter called “learning rate”. Broadly speaking, in machine learning the learning rate plays the same role of the "step length" in first order optimization methods.

First of all, weights and biases of the network are initialized. Then, the input data are fed in the forward direction through the network. The forward propagation of input data generates the outputs. The loss function is used to compute the error with respect to the target. After the forward pass through the network, backpropagation performs a backward pass while adjusting weights and biases. Backpropagation is in essence an efficient application of the chain rule that avoids redoing some of the same computations over and over again [24, 5]. After parameter updating, a new forward propagation is performed, backpropagation follows and so on. The trained model is deterministic.

Several weight initialization methods, loss functions and optimization algorithms exist. Trial and error is used to select good training options that are highly problem dependent. Today, it is common to start training with Glorot initialization and Adam optimizer [24, 28, 35].

The Glorot (also known as Xavier) initialization is the most popular initialization scheme [36]. The simpler method that initializes weights with small uniform random values does not work well with deep networks (see Figure 13).

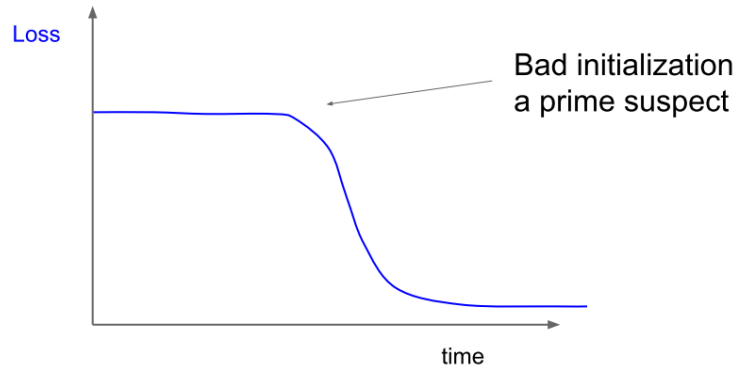

Figure 13: effects of bad initialization (figure taken from [28])

For each layer, Glorot initialization draws samples from a truncated distribution centered on 0 with standard deviation:

$$std = \sqrt{\frac{2}{fan_{in} + fan_{out}}},$$

where fan_{in} and fan_{out} are the number of input and output units in the weight matrix of the layer respectively. The distribution can be normal or uniform.

Adam is an adaptive learning rate optimization algorithm that has been designed specifically for training deep neural networks [29]. It is an algorithm for first-order gradient-based optimization of stochastic objective functions, based on adaptive estimates of lower-order moments. Adam differs from classical stochastic gradient descent [30]. Benefits of using Adam are [29]:

- straightforward implementation;
- computational efficiency;
- little memory requirements;
- invariance to diagonal rescaling of the gradients;
- suitability for problems that are large in terms of data and/or parameters;
- suitability for non-stationary objectives;
- suitability for problems with very noisy/or sparse gradients;
- intuitive interpretation of hyper-parameters that typically require little tuning.

While stochastic gradient descent maintains a single learning rate for all weight updates and the learning rate does not change during training, Adam maintains a special learning rate for each network parameter and separately adapts each one as learning unfolds. Adam is relatively easy to configure because its default configuration parameters are often characterized by a fixed learning rate and work well in training neural networks for most problems.

A training dataset is comprised of many samples (a sample is a single row data that contains inputs and targets).

Since a network can include millions of parameters and it is common to train it using tens or hundreds of thousand data, it is not efficient to compute gradient with respect to all the samples in the training set. It is more convenient and theoretically correct to approach the minimization problem stochastically by

computing the gradient with respect to a subset of samples of the training set (called mini-batch and containing n samples) at each step of the optimization process. The optimization process consists of:

1) extraction, without repetition, of n (or less if there are not enough) samples from the training set that have not been extracted yet;
2) computation of the gradient and everything else needed with respect to the mini-batch (samples extracted) and implementation of one single optimization step;
3) if exist samples in the training set that have not been extracted yet, go to 1); otherwise, stop.

One single pass, with steps (1)-(3), over all the samples in the training set is called epoch and usually a consistent number of training epochs is required in practice to reach the desired performance from the model.
According to Goodfellow et al., the most important hyperparameter to tune is often the learning rate [24]. It has a small positive value and controls how quickly the model is adapted to the problem we want to solve. Smaller learning rates require more training epochs, since each weight update involves small changes. Conversely, larger learning rates result in rapid changes. Too large learning rates cause the model to converge too quickly to a suboptimal solution, whereas too smalls learning rates cause the training process to get stuck (figure 14). For these reasons, Adam default configuration with fixed learning rate is largely preferred, letting the users to avoid the problem of tuning the speed of learning rate decay.

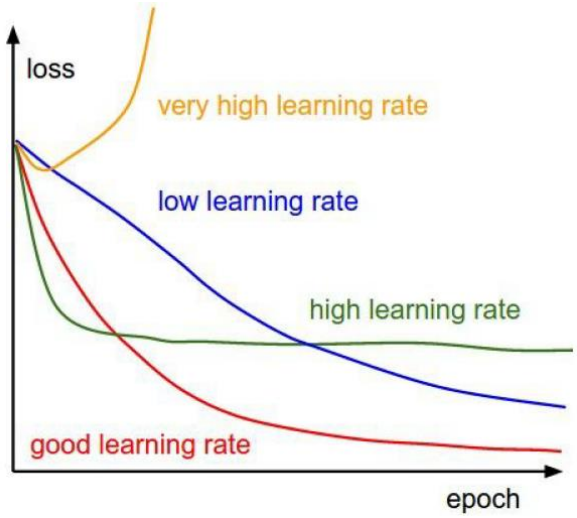

Figure 14: effect of learning rate (figure taken from [28])

Neural network training is not easy because it fails silently [35]. Errors can be very tricky to detect, because the network is often able to work somehow even with errors. Most machine learning models have multiple parts that are each adaptive. If one part is broken, the other parts can adapt and still achieve roughly acceptable performance [24]. Moreover, when a machine learning system performs poorly, it is usually difficult to tell whether the poor performance is intrinsic to the algorithm itself or whether there is a bug in the implementation of the algorithm [24]. In most cases, we do not know a priori what the intended behavior of the algorithm is. In fact, the entire point of using machine learning is that it will discover useful behavior that we were not able to specify ourselves [24].

According to Andrej Karphaty, the first step to training a neural net is to not touch any neural net code at all and instead begin by thoroughly inspecting input data. This step is critical. It deals with scanning through thousands of samples, understanding their distribution and looking for patterns, data imbalances and biases,

writing some simple code to search, filter and sort by whatever we can think of and visualizing their distributions and the outliers along any axis [35].

After data inspection, it is better to pick some simple model that we couldn't possibly have screwed up somehow [35]. At this stage, we should have a good understanding of the dataset and we have the full training and evaluation pipeline working. It is time to improve model performances. The first step is getting a model large enough that it can overfit. Then, we have to regularize it appropriately. If we are not able to reach a low error rate with any model at all that may again indicate some issues, bugs, or misconfigurations. The most important ingredient to successfully perform training is not knowing a wide variety of machine learning techniques. In practice, one can usually do much better with a correct and methodical application of a commonplace algorithm than by sloppily applying obscure algorithms [24].

The following tips from [35] can be useful:

- don't be creative in the early steps of the projects, find the most related papers and copy their simplest architecture that achieves good performance;
- use Adam optimization algorithm;
- add complexities once at a time;
- disable learning rate decays entirely (use a constant learning rate) and eventually tune this at the very end.

Different strategies can be tried to regularize the model. We can increase the training dataset, use data augmentation, use pretrained networks, reduce input dimensionality by selecting less features to be analyzed by the network, decrease the batch size, add dropout, add weight decay penalty, add batch normalization, implement early stopping, make the model larger. Refer to [24] for detailed descriptions of all these strategies. Only batch normalization is briefly introduced here since it is tried in the training phase of the present work (section 4). Batch normalization is a method of adaptive reparameterization, motivated by the difficulty of training very deep models [24]. It acts to standardize only the mean and variance of each unit in order to stabilize learning by allowing the relationships between units and the nonlinear statistics of a single unit to change [24, 28]. Batch normalization layers are usually inserted after fully connected or convolutional layers and before the activation functions. They compute the empirical mean E and the variance var of their inputs independently for each dimension, then normalize data as follow:

$$\widehat{x_k} = \frac{x_k - E(x_k)}{\sqrt{var(x_k)}}.$$

Batch normalization is found to [37, 38]:

- improve the gradient flow through the network;
- allow higher learning rates;
- reduce the strong dependence on weight initialization;
- act as a form of regularization.

Hyperparameter tuning is usually performed by comparing the accuracy of different sets of hyperparameters on training over few epochs. Best models can then be tuned by training over many epochs. Neural networks keep training for unintuitively long time [35].

At the very end of the training process, model ensembles can be used. They are a pretty much guaranteed way to gain 2% of accuracy on any application [24].

Refer to [35, 24, 28, 5] for further details on training strategies and tips. Particularly, tips from [35] really make a difference for the present work.

2.3. Convolutional neural networks

CNNs are a specialized kind of neural networks for processing data that has a known, grid-like topology. They are good for image data since the pixels of an image can be thought as elements of matrices (in case of black and white images) or 3D tensors (for RGB images). While fully connected networks require to convert images into vectors, convolutional networks preserve spatial structure of image data. The typical feature of CNNs is the use of convolution in place of general matrix multiplication in at least one layer [24]. Convolution is a specialized kind of linear operation. In the simplest case of matrix inputs, it is defined as:

$$X(i,j) = (I * \Phi)(i,j) = \sum_m \sum_n I(i+m, j+n)\, \Phi(m,n).$$

where:

- I is the input matrix;
- Φ is the kernel or filter that contains weights to be updated during training;
- X is the activation map that constitutes the output of a convolutional layer.

Each unit in the activation map is connected to a small region in the input matrix having the same size as the filter. This region is known as receptive field. Each unit in the activation map (represented by the grey spot in figure 15) is the result of taking a dot product between the filter and a specific region of the input matrix. Figure 15 shows an example of convolution on an RGB image of size 32x32 pixels so that the input is a matrix of size 32x32x3. A filter of size 5x5 is applied that reduces the size of the output matrix to 28x28x3.

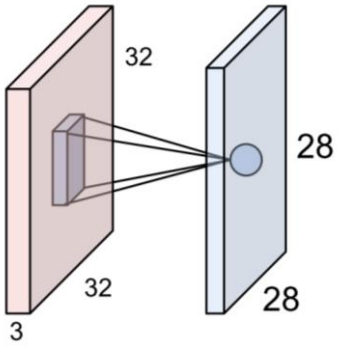

Figure 15: application of a single filter to an RGB image (figure taken from [28])

We can apply more than one filter to the input getting one activation map for each of them. This is shown in figure 16 where 6 filters of size 5x5 are applied to an input matrix of size 32x32x3.

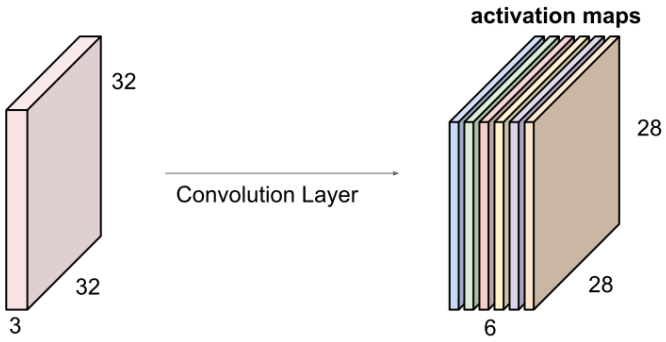

Figure 16: application of multiple filters to an RGB image (figure taken from [28])

Formally, $(I * \Phi)$ is the discrete cross-correlation, but it is usually called convolution in the machine learning community. This is due to the relation between the two operations [24].
Discrete convolution can be viewed as a weight-input matrix multiplication where several learning parameters are constrained to be equal to each other and where the weight matrix is usually very sparse, because the filter is usually smaller than the input image [24]. CNNs are theoretically equivalent to the fully connected ones but characterized by the advantage of having a remarkable smaller number of weights and, therefore, a smaller computational cost. This property is also known as parameter sharing since each weight of the filters is actually shared with more pixels of the inputs.

The typical fundamental structure of a convolutional network consists of:

- several convolutions in parallel that produce a set of linear activations (convolutional layer);
- nonlinear activation functions to make the model able to predict nonlinear effects;

Each convolutional layer requires four hyperparameters:

- the number K of filters;
- the spatial extent F of the filters;
- the stride S;
- the amount of padding P.

The spatial extent F of the filter is the parameter such that $\Phi_1, \dots \Phi_K \in \mathbb{R}^{\{F \times F\}}$, for the 2D case. In case of 3D arrays as inputs, also the depth D_1 of the inputs is required to evaluate the codomain of the filter functions: $\Phi_1, \dots, \Phi_K \in \mathbb{R}^{\{F \times F \times D_1\}}$.

The stride is the number of pixels that shifts over the input matrix at a time. When the stride is 1 then the filters are moved to 1 pixel at a time. When the stride is 2 then they are moved to 2 pixels at a time and so on. Figure 18 shows an example of convolution using 3x3 all-ones filters and a stride of 2.

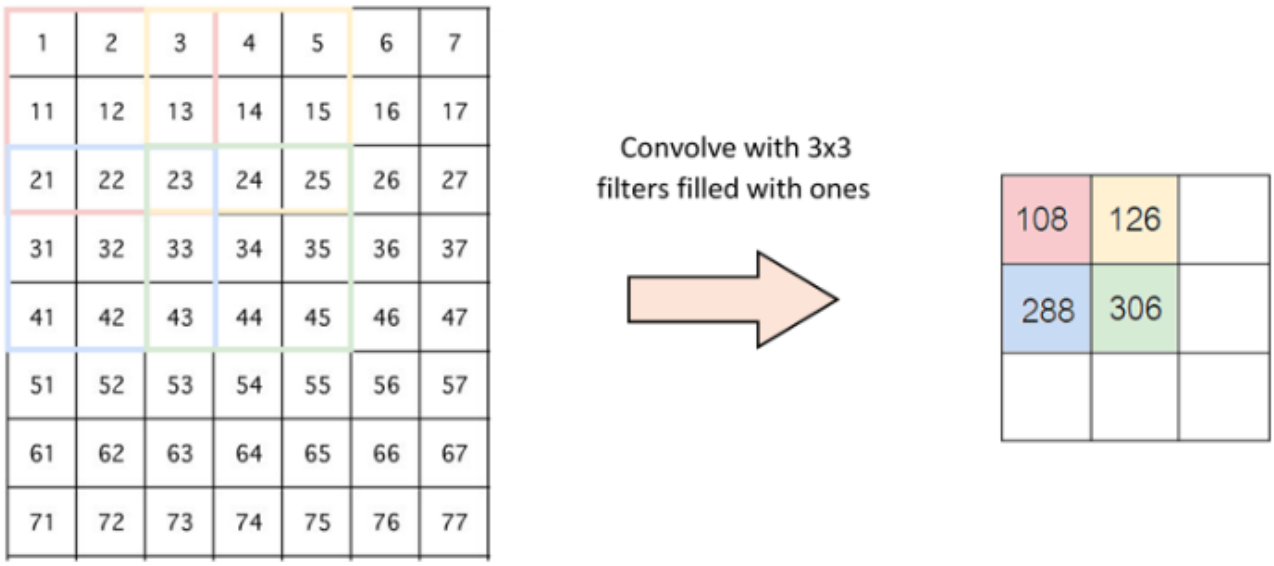

Figure 17: effect of stride on convolution (figure taken from [39])

Padding is needed to perform convolution when the filters do not fit perfectly the input matrix. Two options can be implemented:

- pad the matrix borders with zeros so that the filters fit (zero-padding, see figure 19);

- drop the part of the image where the filters do not fit keeping only the valid part of the image (valid padding).

A convolutional layer accepts a tensor I of size $W_1 \times H_1 \times D_1$ and produces a tensor X of size $W_2 \times H_2 \times D_2$, where:

- $W_2 = (W_1 - F + 2P)/S + 1$;
- $H_2 = (H_1 - F + 2P)/S + 1$;
- $D_2 = K$.

For each $k = 1, \ldots, K$, the k-th array $X_{\{:,:,k\}} \in R^{\{W_2 \times H_2 \times 1\}}$ of the output tensor is obtained from the convolution of I (with padding P) and the k-th filter Φ_k (with stride S). Then, biases are added to the output of the convolution and the activation function is applied. A convolutional layer introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases. Obviously, the stride S and the padding P must be chosen carefully to obtain the desired shape for outputs. For example, a stride $S = 1$ and a zero-padding $P = (F - 1)/2$ can be selected to preserve the input shape. In Figure 19, a case with a 3x3x1 filter and a 7x7x1 image with zero-padding $P = 1$ is shown. By choosing a stride $S = 1$, the output of the convolution will be a tensor of shape 7x7x1.

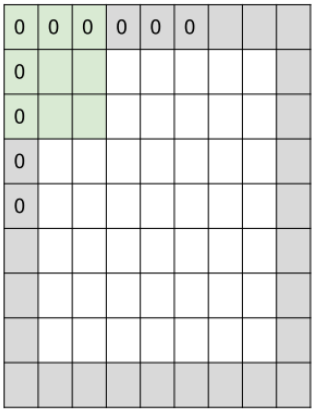

Figure 18: implementation of zero-padding (figure taken from [28])

A special and additional option for convolutional layers is the pooling operation. This operation is usually applied to the activation maps in order to transform the output into a summary and reduced version of it (figure 19).

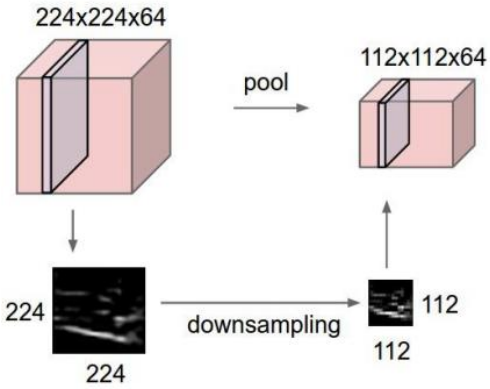

Figure 19: pooling layer (figure taken from [28])

Since the pooling operation is not always applied, sometimes it is considered as a layer (called pooling layer) separated from the convolutional one. Convolutional layers operate over each activation map separately by replacing the output over a neighborhood region with a single value. They act similarly to convolution (restricted to single activation maps) where filter weights are fixed to one and summation is replaced by a pooling function. The most popular pooling functions are the max-pooling and the average-pooling, that use the maximum and the average of the whole neighborhood respectively [24, 28]. An example of max-pooling application is shown in figure 20.

As the convolution, pooling operations are characterized by a spatial extent F' for the filter, a stride S' and a zero-padding parameter P'. Given the output tensor $X \in R^{\{W_2 \times H_2 \times D_2\}}$ of a convolution for each activation map $X_{\{.,.,k\}}$, with $k \in \{1, \ldots, K = D_2\}$, and a pooling operation (e.g. the max-pooling), the returned output tensor $Y \in R^{\{W_3 \times H_3 \times D_2\}}$ of the pooling operation is such that:

$$Y_{\{i,j,k\}} = \max\left(X_{\{i,j,k\}}, \ldots X_{\{i+F'-1, j+F'-1, k\}}\right),$$

where:

- $W_3 = (W_2 - F')/S' + 1$;
- $H_3 = (H_2 - F')/S' + 1$.

Since pooling computes a fixed function of the input, it does not introduce parameters making the representation smaller and more manageable. It is common to use zero-padding for pooling layers [24].

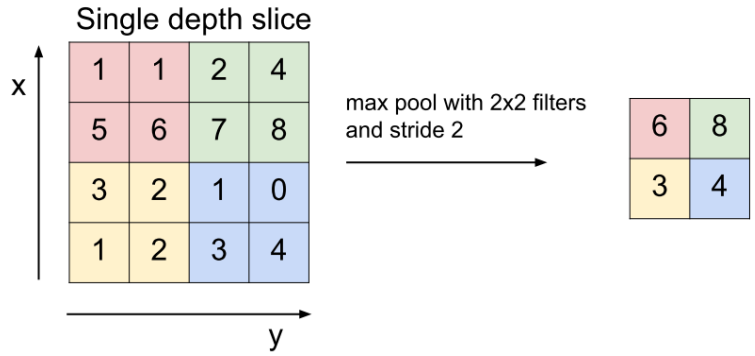

Figure 20: example of max pooling (figure taken from [28])

In convolutional networks, the sequence of convolution, non-linear activation function and pooling is often repeated several times, then some traditional fully connected layers are placed (see figure 21). The idea is to use convolutional layers to extract information from inputs and to process this information through the subsequent fully connected layers [24].

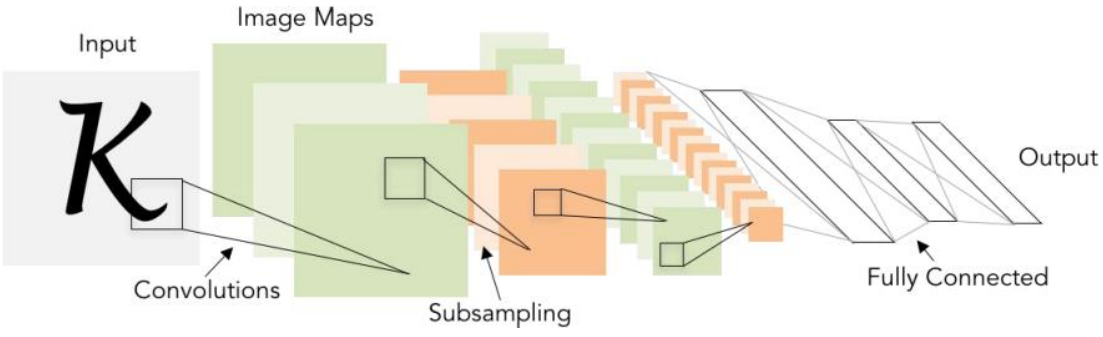

Figure 21:structure of convolutional neural networks (figure taken from [40])

In conclusion, CNNs are characterized by the following features:

- **Sparse interactions** → They are accomplished by making the filter smaller than the input. In this way, we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency by reducing overfitting problems. Performance improvements are usually significant while keeping K several orders of magnitude smaller than W and H [24]. In deep convolutional networks, units in the deeper layers may indirectly interact with a larger portion of the input. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks each describing sparse interactions only.
- **Parameter sharing** → It relates to using the same parameter for more than one function in a model. In convolutional layers, each member of the filter matrix is used at every position of the input (except maybe some of the boundary pixels, depending on the design decisions regarding the boundary). This reduces the storage requirements of the model, so that parameter sharing as well as sparse interaction reduces memory requirements and improves statistical efficiency. The particular form of parameter sharing causes the layer to have equivariance to translation [24].
- **Equivariant representations** → Pooling and parameter sharing help to make the representation approximately invariant to small translations of the input. This property can be very useful if we care more about whether a certain feature is present than exactly where it is. Pooling over spatial regions produces invariance to translation, but if we pool over the outputs of separately parametrized convolutions, the features can learn which transformations to become invariant to [24].
- **Possibility to work with inputs of variable size** → It is achieved using pooling. It can be very useful in a number of practical applications [24].

3. An overview on the application: particle image velocimetry

The rapid evolution of both digital image acquisition and computing capability in the past few decades has determined the development of image-based experimental techniques for the characterization of fluid flows. PIV is an anemometric and non-intrusive technique used in research and industry to provide both qualitative and quantitative information on the instantaneous velocity field of fluid flows. PIV has several advantages compared to other traditional methods such as Hot Wire Anemometry (HWA) or Laser Doppler Velocimetry (LDV). The main advantages are listed in the table below from [41].

HWA, LDV	**PIV**
single point measurements	whole field method
traversing of flow domain	non-intrusive technique
time consuming	instantaneous flow field

Table 1: PIV advantages

3.1. Working principle and experimental set-up

PIV is based on the measurement of the displacement of small tracer particles that are advected by the fluid. This is achieved by comparing two digital images taken with very short time separation (in the order of microseconds).

Since the time delay between the two frames is known, by measuring particle displacements in the i−th direction (Δs_i) it is possible to reconstruct the i−th local velocity component as:

$$V_i = \frac{\Delta s_i}{\Delta t}.$$

The processing algorithm computes the most probable displacement in the image plane of the camera, but we are interested in the target plane where the real displacement is. This is not a problem because image and target planes are strongly related through the magnification factor M_0:

$$\begin{bmatrix}\Delta X \\ \Delta Y\end{bmatrix} = M_0 \begin{bmatrix}\Delta x \\ \Delta y\end{bmatrix},$$

where uppercase and lowercase letters indicate the target and the image plane respectively.

The magnification factor is a dimensionless number given by the ratio of two lengths:

$$M_0 = \frac{z_0}{Z_0},$$

where z_0 is the focal length and Z_0 is the distance between the camera lens and the target plane of the fluid flow.

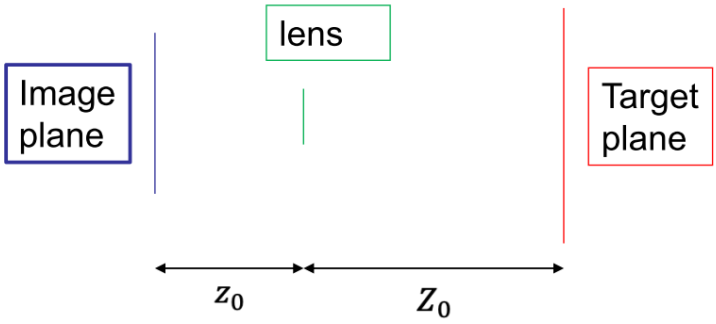

Figure 22: image plane and target plane

We can resolve the two in-plane components of the velocity field (2C2D) by placing the camera with its recording plane parallel to the laser sheet (standard PIV). By placing two cameras with a proper angle with respect to the laser sheet instead (stereoscopic PIV), it is possible to resolve all the three velocity components (3C2D). It is also possible to measure the three components in a 3D volume (3C3D), using two or more cameras (tomographic PIV). Only standard PIV is considered in the present work.

The tracer particles have a scattering effect when illuminated by laser light. This effect makes the flow visible. The flow itself has no scattering effect, so tracer particles are needed to successfully perform PIV. They must be homogeneously distributed in the observed region to obtain robust unbiased measurements over the flow domain. They are illuminated using a thin light sheet. The laser light source produces a high-energy, pulsed, coherent light beam that is shaped into a thin light sheet using a system of optical lens. The light scattered by the tracer particles is recorded at two subsequent times by digital camera. Then, the particle images are processed through a cross-correlation based algorithm to get velocity field maps. The laser pulses and camera acquisition must be synchronized.

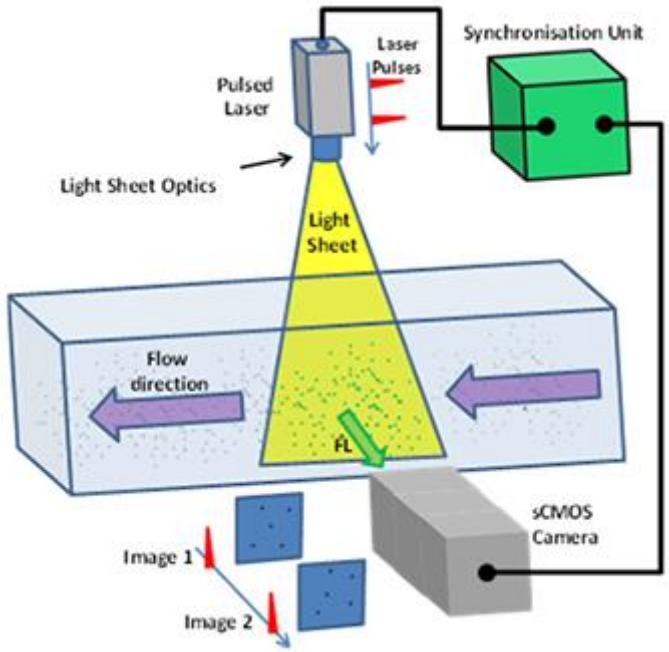

Figure 23: Schematic of the main components of a typical PIV setup (figure taken from [42])

Aside from the tracer particles and the laser light sheet, no perturbation is introduced inside the test section so that PIV is a non-intrusive technique. The tracer particles are used to seed the flow and they must be

sufficiently small in order to follow properly the fluid motion without any appreciable modification to the properties of the flow. The type of tracer particles is chosen by the Stokes number:

$$St = \frac{t_p}{t_0} = \frac{d_p^2 \, \rho_p / \mu}{t_0}$$

where:

- t_0 is the typical time of the flow and t_p is the particle relaxation time;
- d_p is the particle diameter (typical values in air are around μm);
- ρ_p is the particle density;
- μ is the dynamic viscosity of the fluid flow.

Good tracer particles make the Stokes number much smaller than 1.

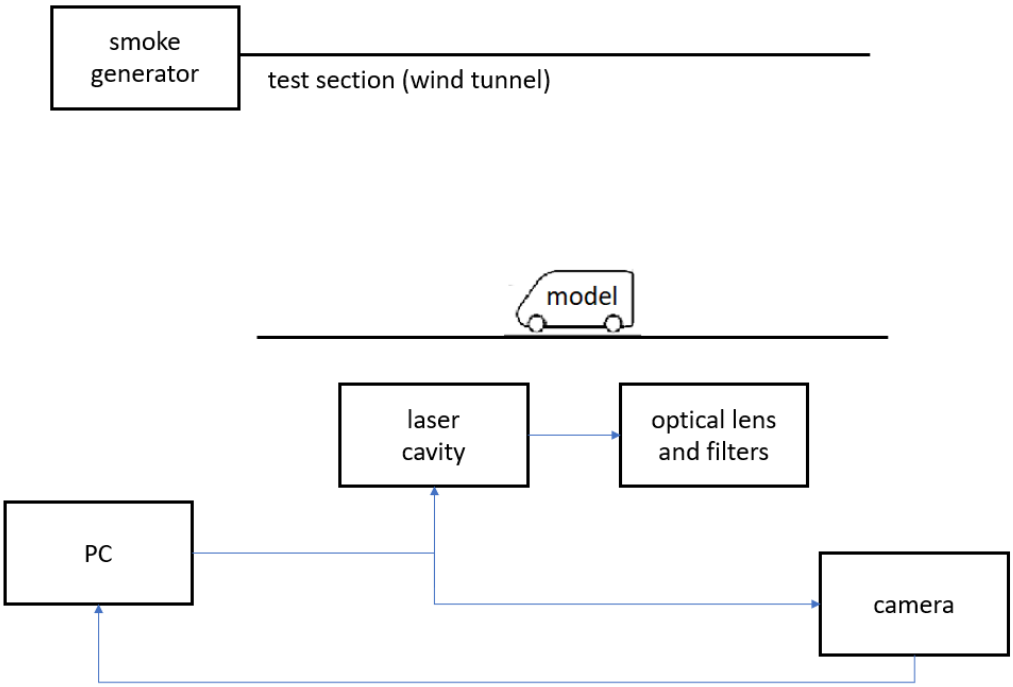

Figure 24: PIV measurement chain

Lasers for PIV applications can be classified depending on the operating frequency:

- Low speed lasers: repetition rates up to 15Hz, energy per pulse up to 800mJ (typically 200mJ);
- High speed lasers: repetition rates up to 10KHz, energy per pulse up to 30mJ.

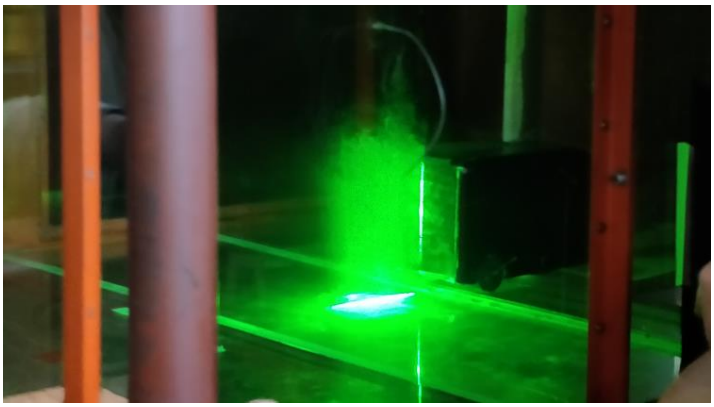

Figure 25: laser positioning, PIV experiment on a vehicle wake

PIV cameras are differentiated depending on the sensor technology:

- CCD (Charge-Coupled Device): low price, low brightness (up to 12bit), higher probability of damaging the sensor;
- CMOS (Complementary Metal-Oxide Semiconductor): higher price, much brighter (up to 16bit), very reliable.

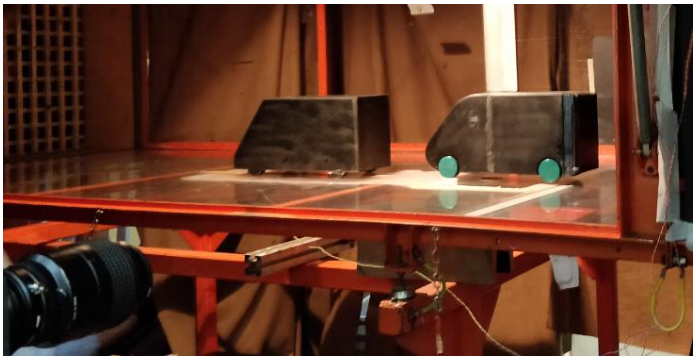

Figure 26: camera positioning, PIV experiment on a vehicle wake

For further descriptions on advanced PIV principle refer to [12, 14].

3.2. Standard processing methods

The standard processing method to perform PIV is based on cross-correlation [43, 44, 45]. The data domain is split up in small interrogation windows and the cross-correlation is computed on each of them. One velocity vector is found for each window using this method.

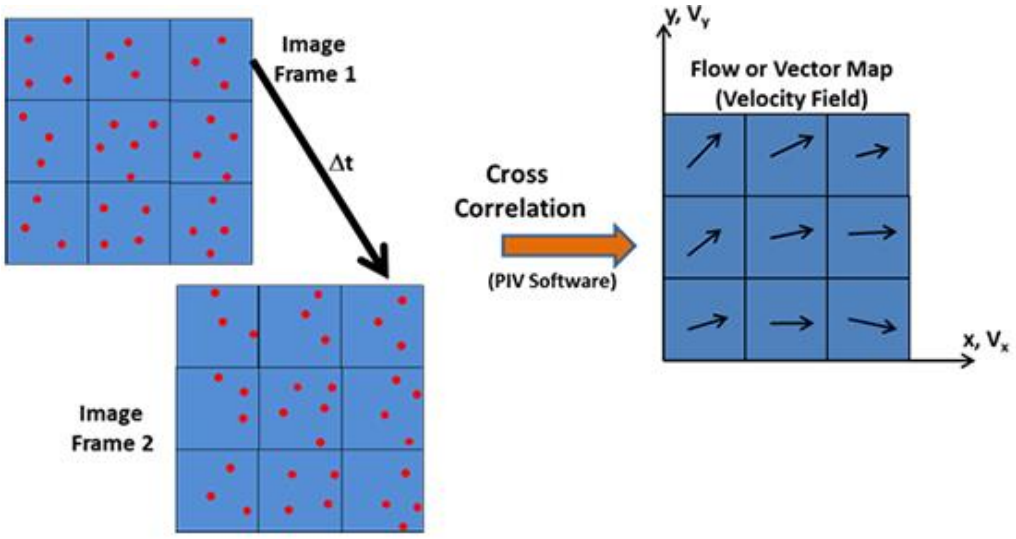

Figure 27: process of reconstruction of the velocity field via cross-correlation (figure taken from [42])

The correlation of two functions $f(x)$ and $g(x)$ is defined as the integral of the product of $f^*(x)$ with $g(x)$, the latter shifted over some distance Δx.

$$f * g = C_{f_g}(\Delta x) = \int_{-\infty}^{+\infty} f^*(x) g(x + \Delta x)\, dx$$

where the asterisk indicates the complex conjugated value. The asterisk has no function in the PIV case, since real signals are considered. The correlation is computed for every shift Δx. If the observed structure is moving as a whole, a correlation maximum will be found on the real shift, besides random and offset correlations.

If the number of points per function is N, the evaluation of the correlation involves N^2 operations, since the correlation of two discretized functions must be calculated. If this is performed in a two-dimensional plane of two $N \times N$ collocation points, the number of operations increases to N^4, so the number of operations is usually very large. This is the reason why Fourier theory is employed to evaluate correlations. If the Fourier transforms on discrete sets of data are performed with Fast Fourier Transform (FFT), then the correlation involves only $O(N \log_2 N)$ operations for a one-dimensional case and $O(2N^2 \log_2 N)$ for a two-dimensional correlation [13, 46]. This is considerably less then $O(N^4)$ for reasonable values of N and so processing via cross-correlation is performed by computing two-dimensional FFT on both interrogation areas separately. The cross-correlation is evaluated in the frequency domain by complex conjugate multiplication of the transforms. Results are then back transformed to the physical domain.

The output of the cross-correlation algorithm is the correlation factor ϕ_{im}:

$$\phi_{im} = \frac{\sum_{i,j}^{W}\left(\left(f(i,j) - \mu_f\right)\left(g(i+l, j+m) - \mu_g\right)\right)}{\sqrt{\sum_{i,j}^{W}\left(f(i,j) - \mu_f\right)^2 \sum_{i,j}^{W}\left(g(i+l, j+m) - \mu_g\right)^2}}$$

where:

- f and g are related to the first and second particle image respectively;
- μ_f and μ_g are the mean values of particle density of the two images (better results can be achieved if mean values are subtracted before performing cross-correlation);
- i and j indicate the dimensions of the domain;

- l and m indicate displacement in i and j direction.

Typical peak values of ϕ_{im} are around $0.6 \div 0.7$ and it is difficult to obtain higher values because of image noise and other problems that are mentioned below. A number of pre-processing techniques can be used to improve the quality of the velocity field estimations.

The main problems encountered when performing cross correlations in PIV experiments are listed below:

- **Edges** → Particle pairs near edges of the interrogation windows contribute less to correlation. This is why overlap between adjacent interrogation windows can be allowed when splitting up the domain. Overlap is commonly applied, but too much overlap causes oversampling [41].
- **Background noise** → High-pass filtering can be used to eliminate the background intensity variations. It should be used with caution, because can slightly affect data. If the input data has a reasonable quality and the displacements are small enough, then straightforward cross-correlation processing gives accurate results [13]. However, high-pass filtering is commonly used to improve cross correlation results.
- **Random correlations** → Random correlations originate from the correlation of different particles, the correlations of particles with background and noise intensity and correlations involving particles illuminated only once because they are entering and leaving the domain due to out-of-plane motion.
- **Particle loss** → There are incomplete particle pairs due to the in-plane entering and leaving of particles in the interrogation area. Since particles are moving with the flow, it may happen that a particle imaged in the interrogation area in the first frame is advected over the boundary of it in the second one. Conversely, particles may be advected into the interrogation area in the time interval between frame recordings. This phenomenon is called in-plane particle loss. Most of the particles are imaged two times in the interrogation area with low velocities, so that the in-plane particle loss is small. However, the in-plane particle loss increases with velocity and causes the correlation to decrease. Some correction techniques can be applied to the correlation to consider particle loss effects, but the dimensions of the interrogation windows must be sufficiently large to include in the cross-correlation process a relatively large number of particles. This means that a relatively low number of velocity vectors can be computed so that poor spatial resolution is achieved while determining the velocity field.

Typically, the number of particles in each interrogation window have to be in the range of $10 \div 15$ to ensure high correlation coefficient. Both higher and lower values of particle density cause lower values of the correlation coefficient so that the spatial resolution is strongly dependent on the image density.

PIV results also highly depend on the temporal separation between the two image frames. Usually, the temporal separation is chosen to have displacement of tracer particles between 8 and 12 pixels. Higher separations (and so higher displacement) allow to investigate higher dynamic ranges, but they make the correlation peak decrease. On the other hand, lower temporal separations make the error increase and the dynamic range decrease.

Each time a new velocity vector is computed, data validation criterions must be applied. This is done at each intermediate step of the iteration process.

4. Velocity field regression with deep learning

The network architecture is built reflecting the model proposed by Rabault et al. [15], then training is performed. Synthetic image data are generated to feed the network. Hyperparameters are progressively set by trial and error using a validation dataset. Partial changes of the initial architecture and of the data generation process are evaluated. The best trained models are firstly tested on images generated independently but in the same way as the training dataset. If the achieved results are considered sufficiently good, a second test is performed on images generated with a different process using benchmark velocity fields. Then, the trained models are tested on synthetic images by the first PIV International Challenge [48] and real images. While 32x32 black and white pixel images are used for the first test as well as for the training phase, the other tests are done on larger images. Each large image is split into windows of 32x32 pixels that are processed by the trained network. Performances are evaluated using the following quantities:

- The Frobenius norm of the error:

$$FN = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left[\left(U_i - \widehat{U_i}\right)^2 + \left(V_i - \widehat{V_i}\right)^2\right]},$$

where (U_i, V_i) and $(\widehat{U_i}, \widehat{V_i})$ are the velocity prediction and the true velocity field on the i-th pixel respectively and N is the total number of pixels. (U_i, V_i) can be the mean of several predictions obtained by feeding the network with different particle images of the same velocity field or the mean of predictions from different networks or both.

- The RMSE error on each velocity component:

$$RMSE(X) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(X_i - \widehat{X_i}\right)^2},$$

where X is U or V.

- The bias error on each velocity component:

$$bias(X) = \frac{1}{N}\sum_{i=1}^{N}\left(X_i - \widehat{X_i}\right)^2,$$

where X is U or V.

Absolute and relative errors are also computed in the first test for monitoring purposes. Matlab is used for the whole simulation process.

4.1. Hardware and software

Matlab 2019b with Deep Learning Toolbox is used to implement the neural network model and all the algorithms needed for the training and the test phase.

The particle image generator is written in Matlab 2019b using the Image Processing Toolbox and it works also on Matlab 2019a. Moreover, it is supposed to work on all Matlab older versions where the Image Processing Toolbox can be installed. On the contrary, Matlab 2019b release is required to build the neural

network model and perform training and tests. Older versions of Matlab are not enough even if they support the Deep Learning Toolbox, because a major update is introduced in Matlab 2019b dealing with this Toolbox. Particularly, it concerns the training flexibility and the possibility to implement custom functionalities: train advanced network architectures using custom training loops, automatic differentiation, shared weights, custom loss functions and multiple data normalization options. The final version of the model can run on Matlab 2019b only, because of the use of PReLU and custom loss function.

The Parallel Computing Toolbox is required to speed up training on a single or multiple GPU workstation. Training is theoretically possible even on single CPU, but this option is so slow that makes impossible to train the model, especially if a large number of simulations have to be launched to select the best set of training parameters and network architecture layers by trial and error.

It should be noted that GPU acceleration through Parallel Computing Toolbox requires a GPU that supports CUDA 3 or newer. This means that a Nvidia GPU is needed, since CUDA is a parallel computing platform created by Nvidia and it is supported by nVidia GPUs only.

The present work is developed using a nVidia Tesla V100 SXM2 - 32 GB - 5120 CUDA cores located on the High Performance Computer (HPC) by HPC@POLITO [47]. Matlab codes are written on a local PC, then both the generation of training images and the training phase are performed remotely on the HPC. The trained network is downloaded and tested locally. Particle images for the test phase are generated locally.

The remote PC has the following specifications:

Architecture	Linux Infiniband-EDR MIMD Distributed Shared-Memory Cluster
Node Interconnect	Infiniband EDR 100 Gb/s
Service Network	Gigabit Ethernet 1 Gb/s
CPU Model	2x Intel Xeon Scalable Processors Gold 6130 2.10 GHz 16 cores
GPU Model	8x nVidia Tesla V100 SXM2 - 32 GB - 5120 cuda cores
Sustained performance (Rmax)	21.094 TFLOPS
Peak performance (Rpeak)	27.238 TFLOPS
Computing Cores	448
Number of Nodes	14
Total RAM Memory	5.376 TB DDR4 REGISTERED ECC
OS	Centos 7 - OpenHPC 1.3
Scheduler	SLURM

Table 2: specification of remote HPC

The local PC has the following specifications:

Processor	AMD A8-7410 APU, 2.20 GHz, 4 logical processors with AMD Radeon R5 Graphics
RAM	8 GB
Disk	HDD
Operating System	Windows 10
Architecture	64-bit

Table 3: specifications of local PC

Note that it is fundamental to save data in the same HPC node where the GPU is located. Training is verified to progress very slow when data are stored in a different node (login node for example) due to the large number of times the training script reads data from folders (once per processed batch).

Matlab is chosen as deep learning framework because it is easier to set than open-source deep learning libraries as TensorFlow that require more skills in computer science. For example, it is almost immediate to launch Matlab simulations from every HPC computational node. Conversely, Python often requires making a container with all the libraries you want to use such as TensorFlow. Moreover, both setting the GPU and launching Python from a node other than the login node require further programming skills.

4.2. Synthetic training dataset

Synthetic data are generated to provide an arbitrarily large training set. Moreover, they allow to work with true velocity fields that are accurately known.

The generation procedure of particle images (figure 30) is similar to what is recommended in the literature [12]. The mathematical structure of velocity fields is taken from Rabault et al. [15]: second order polynomials are created for the u and v components of the velocity. The resulting velocity fields represent 2-dimensional and stationary motion:

$$\begin{cases} u = u_0 + \overline{J_u} dr + dr^T \overline{\overline{H_u}} dr \\ v = v_0 + \overline{J_v} dr + dr^T \overline{\overline{H_v}} dr \end{cases}$$

where:

- $\overline{J_u}$ and $\overline{J_v}$ represent the gradients of the horizontal and vertical velocity components respectively;
- $\overline{\overline{H_u}}$ and $\overline{\overline{H_v}}$ represent the Hessian matrices related to the horizontal and vertical velocity components respectively.

Since:

$$\overline{J_i} = [J_{i_{11}} \quad J_{i_{12}}] = \left[\frac{\partial i}{\partial x} \quad \frac{\partial i}{\partial y}\right],$$

$$\overline{\overline{H_i}} = \begin{bmatrix} H_{i_{11}} & H_{i_{21}} \\ H_{i_{21}} & H_{i_{22}} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial^2 i}{\partial x^2} & \dfrac{\partial^2 i}{\partial x\, \partial y} \\ \dfrac{\partial^2 i}{\partial y\, \partial x} & \dfrac{\partial^2 i}{\partial y^2} \end{bmatrix},$$

where $i = \{u, v\}$, then the velocity field can be rewritten as follow:

$$\begin{cases} u = u_0 + [J_{u_{11}} \quad J_{u_{12}}] \begin{bmatrix} dx \\ dy \end{bmatrix} + [dx \quad dy] \begin{bmatrix} H_{u_{11}} & H_{u_{21}} \\ H_{u_{21}} & H_{u_{22}} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} \\ v = v_0 + [J_{v_{11}} \quad J_{v_{12}}] \begin{bmatrix} dx \\ dy \end{bmatrix} + [dx \quad dy] \begin{bmatrix} H_{v_{11}} & H_{v_{21}} \\ H_{v_{21}} & H_{v_{22}} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} \end{cases} \Rightarrow$$

$$\Rightarrow \begin{cases} u = u_0 + \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} + \begin{bmatrix} dx & dy \end{bmatrix} \begin{bmatrix} \frac{\partial^2 u}{\partial x^2} & \frac{\partial^2 u}{\partial x\, \partial y} \\ \frac{\partial^2 u}{\partial y\, \partial x} & \frac{\partial^2 u}{\partial y^2} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} \\ v = v_0 + \begin{bmatrix} \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} + \begin{bmatrix} dx & dy \end{bmatrix} \begin{bmatrix} \frac{\partial^2 v}{\partial x^2} & \frac{\partial^2 v}{\partial x\, \partial y} \\ \frac{\partial^2 v}{\partial y\, \partial x} & \frac{\partial^2 v}{\partial y^2} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} \end{cases}.$$

The origin of the reference system is located at the center of the particle image and the axes are parallel to the image edges, so that u_0 and v_0 correspond to the velocity field at the center of the image (figure 28).

Each time a new image pair is generated, new values are randomly assigned to the coefficients of the second order polynomial so that each image pair is generated independently from the rest of the dataset.

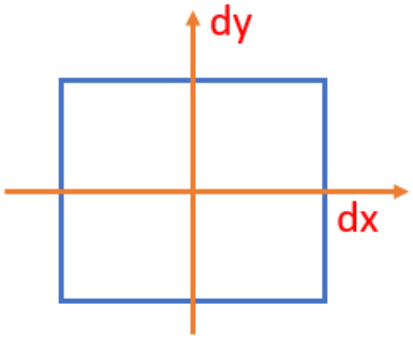

Figure 28: reference system for particle images

For each image pair, constant and gradient coefficients ($u_0, v_0, J_{u_{ij}}, J_{v_{ij}}$) are passed as target to the training model. As a first attempt, numerical ranges for random distributions are taken from Rabault et al. [15], then they are modified to test neural networks on more realistic synthetic images (table 4). Once the random velocity field is generated, a random uniform distribution of material points representing the particle centers is placed in a matrix of size $[xmax, ymax]$ that is slightly larger than the resolution of the images $[resx, resy]$. This distribution is integrated half a time-step backward and forward in time to determine the position of the particle centers in both the frames of the image pair. Integration corresponds to the multiplication for the time step, since the velocity fields are constant in time between the first and the second frame.

Sufficiently small particles recorded by a camera form circular patterns known as Airy disks. The main lobe of diffraction is normally well approximated by a Gaussian bell curve. Gaussian distributions are therefore used to generate the synthetic particle images [12]:

$$I = I_0 \exp\left(\frac{-(x - x_0)^2 - (y - y_0)^2}{\left(\frac{1/8}{d_\tau^2}\right)} \right),$$

where:

- I_0 is the particle image luminosity;
- (x_0, y_0) indicates the position of each particle center;
- d_τ is the effective diameter of the particles in pixels.

Gaussian white noise of variance 1% of the maximum image intensity is added to the images to perform training on non-perfect data producing a more robust model. After adding noise, a threshold of 255 is imposed on image luminosity: eventual higher values originating by the overlap of different particles are set to 255. Luminosity values can be either normalized by 255 or not normalized, so that two different image formats are tested:

- image luminosity in the range of 0-255, natural numbers;
- image luminosity in the range of 0-1, real numbers.

Finally, images are cropped from $[xmax, ymax]$ to $[resx, resy]$ to reproduce situations in which particles leave and enter the image domain between the first and the second frame. An example of synthetic particle images is shown in figure 29.

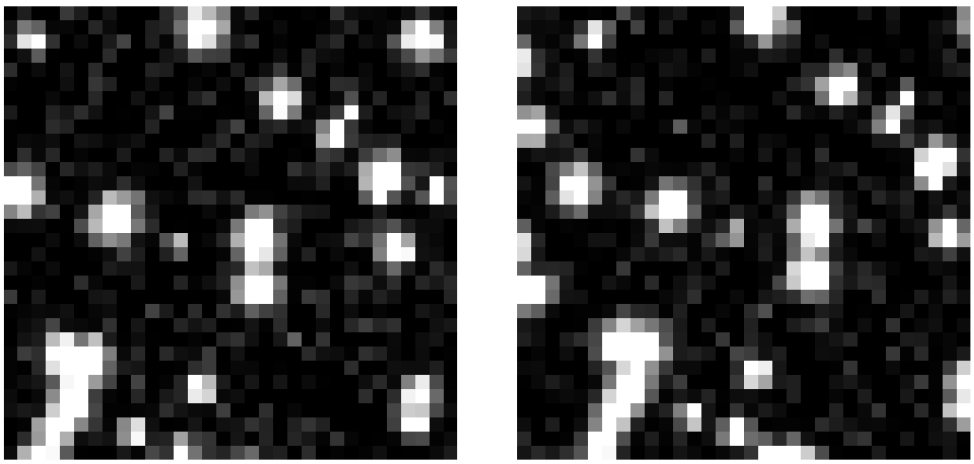

Figure 29: example of particle images, size [32x32]: first exposition (sx) and second exposition (dx)

The number of decimals is set to 4 for both the velocity field and the particle images.

VELOCITY FIELD	**first attempt value**	**changes**	**measure**
$\boldsymbol{u_0, v_0}$	[-4, 4] (uniform)	[-10, 10] (uniform)	px / frame
$\boldsymbol{J_u, J_v}$	[-0.05, 0.05] (uniform)	-	1 /frame
$\boldsymbol{H_u, H_v}$	[-0.005, 0.005] (uniform)	-	1 / (px · frame)
PARTICLES	**first attempt value**	**changes**	**measure**
particle density	0.029	0.8	ppp
particle image diameter	[1.5, 4]	-	px
peak intensity	[200, 255]	-	grey value
IMAGE	**first attempt value**	**changes**	**measure**
noise	max intensity /100	-	grey value
intensity normalization	[0, 1] (real)	[0, 255] (integer)	grey value
decimals	4	15, 2	-

Table 4: numerical values for image generation

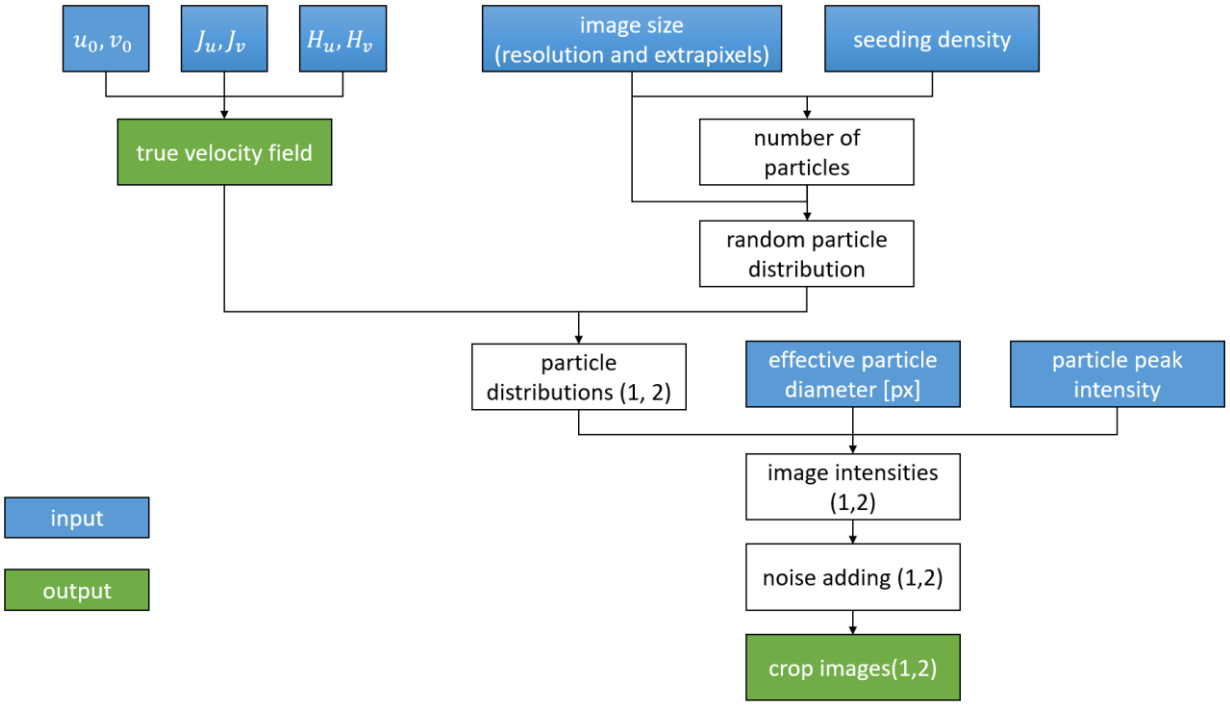

Figure 30: generation process of particle images

4.3. Network architecture

The main reference to build the network architecture is the best prototype presented by Rabault et al. [15]. It is a CNN consisting of a single convolutional layer followed by 4 fully connected layers. Input data are numeric matrices of size 32x32x2: each matrix includes 2 grayscale particle images as two channels (they correspond to the first and second exposition of a standard PIV experiment with stationary 2D motion, see section 3.1 and 4.2). The convolutional layer features 512 kernels of size 16x16 pixels and depth 2 applied with a stride of 8 pixels, so that the size out of the convolutional layer is 8192. The fully connected layers contain 8192, 4096, 2048 and 6 neurons respectively going downwards in the network. All layers use leaky ReLU with scale 0.1 as activation function, except the last. The last layer is followed by a ReLU activation function instead. The output of the network is a vector of length 6 containing predictions for u_0, v_0, J_u, J_v of the input pair of images.

The following two changes are applied to improve the performances of the network (the final version of the network is shown in figure 31):

- an average pooling layer of size 5x5 and stride 1 is added after the convolutional layer;
- the ReLU activation function is replaced with the PReLU one (section 2.2.1).

They have been selected using a trial and error procedure.

Both max and average pooling were tested, the average pooling layer was found to work slightly better than the other one. Similarly, PReLU activation function was found to work slightly better than ReLU when the network was trained using L2 regularization.

Batch normalization (section 2.2.3) after each single layer or all layers was tested, but it was not included in the final version of the model because it reduced the performances of the network.

	Name	Type	Activations	Learnables
1	imageinput 32x32x2 images with 'zerocenter' normalization	Image Input	32×32×2	-
2	conv 512 16x16x2 convolutions with stride [8 8] and padding 'same'	Convolution	4×4×512	Weights 16×16×2×512 Bias 1×1×512
3	leakyrelu_1 Leaky ReLU with scale 0.1	Leaky ReLU	4×4×512	-
4	avgpool2d 5x5 average pooling with stride [1 1] and padding 'same'	Average Pooling	4×4×512	-
5	leakyrelu_2 Leaky ReLU with scale 0.1	Leaky ReLU	4×4×512	-
6	fc_1 8192 fully connected layer	Fully Connected	1×1×8192	Weights 8192×8192 Bias 8192×1
7	leakyrelu_3 Leaky ReLU with scale 0.1	Leaky ReLU	1×1×8192	-
8	fc_2 4096 fully connected layer	Fully Connected	1×1×4096	Weights 4096×8192 Bias 4096×1
9	leakyrelu_4 Leaky ReLU with scale 0.1	Leaky ReLU	1×1×4096	-
10	fc_3 2048 fully connected layer	Fully Connected	1×1×2048	Weights 2048×4096 Bias 2048×1
11	prelu PReLU with 2048 channels	preluLayer	1×1×2048	Alpha 1×1×2048
12	fc_4 6 fully connected layer	Fully Connected	1×1×6	Weights 6×2048 Bias 6×1
13	output absolute error	Regression Output	-	-

Figure 31: final network architecture from Deep Network Designer (Matlab)

4.4. Training

The whole training process is performed using Matlab and GPU acceleration via Parallel Computing Toolbox (section 4.1). Matlab datastore function is used to collect particle image data and true velocity fields. In this way, it is possible to work with data that are too large to fit in memory. Datastores allow to read and analyze data from each file in smaller portions that singly do fit in memory. In the present work, each portion corresponds to a training batch. A file with mat extension is created for each input and each target, so that each file contains only 1 variable of size 32x32x2 (if input) or 1x6 (if target). Transformed and combined datastores are used to match data with Matlab Deep Learning Toolbox formats.

Rabault et al. apply Adam solver and absolute error loss function [15]. They also use a batch size of 128 and initialize weights via Glorot normal distribution [15]. No other information about training options is given as reference. Trial and error method is applied to determine the total number of training and validation data, the learning rate, the need and factor of L2 regularization and the total number of epochs. The same procedure is also applied to verify benefits of using the available reference options, since they may not match properly with the other options selected by trial and error that could be different from the reference work.

Adam solver (section 2.2.3), absolute error loss function and batch size are included in the final model. Other solver such as Stochastic Gradient Descent with Momentum (SGDM) and Root Mean Square Propagation (RMSProp) are tested to be underperforming. It should be noted that Adam is considered the default choice to train a new model because it is relatively easy to configure on most problems where the default configuration parameters do well [24, 28].

Weight initialization via Glorot uniform distribution is adopted, because it worked slightly better than Glorot normal distribution during the training phase.

The adopted loss function is:

$$\sum_{batch\ size} \sum_{output\ size} |p_i - t_i|,$$

where:

- p and t indicate predictions and targets of the network respectively;
- the output size is 6 (it is the number of the outputs of the network).

It is found to perform much better than the mean absolute error that doesn't even allow training to start.

A validation set is used to monitor underfitting and overfitting, progressively setting training hyperparameters by trial and error. Since Adam has an internal decay rate, the base learning rate is set to be constant over epochs.

Parameter	Choice	Other attempts
Solver	Adam	sgdm rmsprop
Weight initialization	Glorot uniform	Glorot normal zeros ones he narrow-normal
Base learning rate	1e-05	1e-04 5e-05 5e-05 for the first half of epochs, then 1e-05
Training dataset	9/10	19/20
Validation dataset	1/10	1/20
Batch size	128	64 256
Loss function	absolute error	mean absolute error absolute error + L2 norm of gradients
L2 regularization	0.05	0.5 0.05 0.005 0.0005

Table 5: training options and network architecture, choices

Several simulations are compared using the root mean square error (RMSE) on the absolute error of the outputs after 100 training epochs:

$$RMSE = \sqrt{\sum_{i=1}^{output\ size} mean(t_i - p_i)},$$

where $mean$ indicates the mean over the whole validation set.

Absolute and relative errors were also computed on every output after 100 epochs in the first phases of the training process:

$$absolute\ error = |prediction - target|,$$

$$relative\ error = \frac{|prediction - target|}{|target|}.$$

RMSE is convenient to monitor training on Matlab because it is automatically computed by Deep Learning Toolbox during the whole training process on both validation sets and batches. So, both the final value of RMSE and the trend over epochs can be rapidly evaluated. Further training is performed on cases when the RMSE appears far from convergence after 100 epochs.

Results of the main simulations are presented in the tables 6 and 7. Table 6 deals with changes in training options and network architecture, while table 7 is about changes in the generation process of synthetic images. Note that the best attempts are only achieved using the set of options shown in table 5. Comparisons described in tables 6 and 7 make sense only for attempts sharing the tested feature, attempts with different tested features may differ for more than one feature.

Tested feature	**Attempts**	**Final RMSE**	
		100 epochs	200 epochs
Number of training data	10 000 100 000 300 000 400 000 500 000 1 000 000 2 000 000	0.44 0.22 0.14 0.12 0.12 0.10 0.09	
Learning rate	1e-04 (constant) 5e-05 (constant) 1e-05 (constant) 5e-06 1e-06 (constant) 5e-05 → 1e-05	0.14 0.14 0.12 0.14 0.18 0.14	
Loss function	absolute error mean absolute error absolute error + L2 norm of gradients	0.12 3.27 0.14	
Pooling	No maximum 5x5 average 5x5 average 3x3 average 6x6	0.15 0.16 0.12 0.22 0.52	
Batch normalization **Batch normalization**	no after convolution	0.12 0.12	 0.12

Tested feature	Attempts	Final RMSE	
		100 epochs	200 epochs
	after average pooling	0.18	
	after first fully connected layer	0.15	
	after last fully connected layer	0.15	
	after each layer	0.19	
Batch size	64	0.15	
	128	0.12	
	256	0.12	
Dropout	L2 without dropout	0.12	
	before last fully connected layer (no L2)	0.15	
	before last fully connected layer (with L2)	0.13	
	before third fully connected layer (no L2)	0.14	0.13
L2 regularization	no	0.13	
	0.5	0.18	
	0.05	0.12	
	0.005	0.11	
	0.0005	0.11	0.12 (overfitting)
Weight initialization	narrow-normal	0.16	
	Glorot uniform	0.12	
	Glorot normal	0.13	
	zeros	3.25	
	ones	3.25	
Target number and type of components	6, constant + gradient	0.12	
	14, constant + gradient + curvature	0.40	
	7, constant + gradient + constant curvature	0.37	

Table 6: training options and network architecture, attempts

Tested feature	Attempts	Final RMSE	
		100 epochs	200 epochs
Particle density [ppp]	0.029 and 400 000 data	0.12	
	0.8 and 400 000 data	0.30	
	0.8 and 1 000 000 data	0.22	
Number of decimals	2 (constant velocity fields)	0.09	
	4	0.12	
	15 (Matlab precision)	0.12	
Velocity range (u_0, v_0)	-4 ÷ 4 and 400 000 data	0.12	
	-10 ÷ 10 and 400 000 data	0.20	0.18
	-10 ÷ 10 and 500 000 data	0.17	
	-10 ÷ 10 and 1 000 000 data	0.14	
	-10 ÷ 10 and 2 000 000 data	0.10	

Tested feature	Attempts	Final RMSE	
		100 epochs	200 epochs
Velocity field components	constant + gradient + curvature (400 000) constant + gradient + curvature (1 000 000) constant + gradient (400 000) constant + gradient (1 000 000)	0.12 0.09 0.08 0.06	 0.10 0.06 (with improvements on gradients)

Table 7: image generator parameters, attempts

After many attempts, the options presented in table 5 are used to train the same network architecture on different tasks (described below) varying the total number of epochs and the size of the dataset. Six trained cases (table 8) are selected to be tested on similar-training images (test 1) and eventually on benchmark fields (test 2). From a rigorous mathematical point of view, they correspond to the same network trained in different ways. However, in the following they are referred as different "networks" or different "models" or Net 1, …, Net 6 for simplicity of notation.

Net	Training fields	Dataset	Epochs	Final RMSE
1	without curvature	1 000 000	200	0.06
2	without curvature	400 000	100	0.08
3	without curvature	1 000 000	100	0.06
4	complete	400 000	100	0.09
5	complete	400 000	100	0.12
6	without curvature + complete	1 000 000	100 + 100	0.09

Table 8: networks for test 1

Nets 4 and 5 of table 8 are trained on particle images generated from the complete velocity field:

$$\begin{cases} u = u_0 + \overline{J_u}dr + dr^T\overline{\overline{H_u}}dr \\ v = v_0 + \overline{J_v}dr + dr^T\overline{\overline{H_v}}dr \end{cases}.$$

Nets 1, 2 and 3 of table 8 are trained on simplified velocity fields without curvature:

$$\begin{cases} u = u_0 + \overline{J_u}dr \\ v = v_0 + \overline{J_v}dr \end{cases}.$$

Net 6 of table 8 is pretrained on simplified velocity fields, then on complete fields.

Note that all the models are trained to predict the simplified velocity field. The difference among them deals only with the particle images used as inputs. In case where complete fields are used, the curvature of velocity fields is a sort of noise on input images. The network is expected to learn to distinguish it from constant motion and gradient effect. In the second case, the noise due to the curvature effect is removed so that it is easier for the network to quantify constant motion and gradient effect during training.

4.5. Benchmark fields

Two benchmark fields are used to further test models of table 8 that give good results on test 1.

The first benchmark field is the Rankine vortex. It is chosen because has a strong curvature effect:

$$V = \begin{cases} v_0 \cdot r/r_0, & r < r_0 \\ v_0 \cdot \dfrac{r_0}{r}, & r > r_0 \end{cases},$$

where:

- v_0 is the maximum velocity;
- r is the distance from the vortex center;
- r_0 is a parameter called core radius (see figure 32).

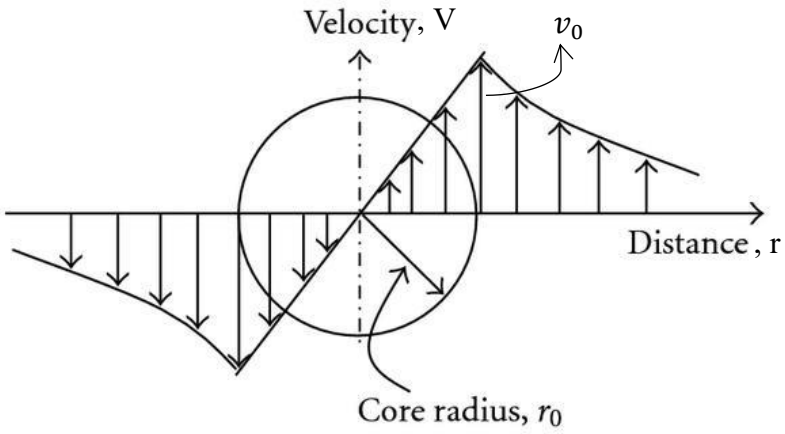

Figure 32: Rankine vortex

The second benchmark is computed by adding to the first one the following shear effect (figure 33):

$$V = (u, v) = \left(\frac{\partial u}{\partial y} y, 0\right)$$

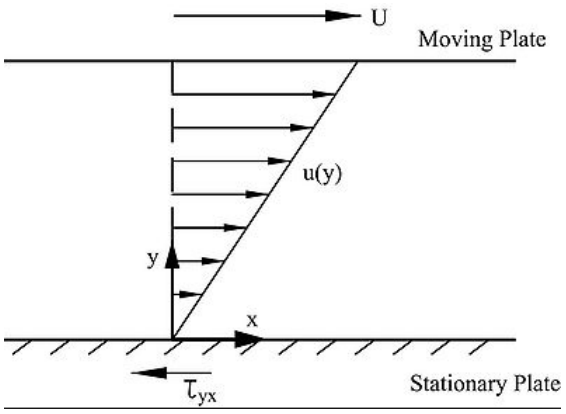

Figure 33: shear flow

It allows to simulate something more similar to boundary layer fields.

4.6. Results

4.6.1. Test 1: small synthetic particle images

This test is performed on particle images of size 32x32 pixels generated with the same process used for training images (constant part + gradient effects + curvature effects). The test aims to determine:

- the best number of particle image pairs related to the same true field to compute the predicted field by averaging (trade-off between accuracy and computational cost);
- the model that minimizes the error.

First, 1000 particle image pairs are produced using the same random velocity field. The predicted field is computed as the mean field of a group of particle image pairs. The convergence of the Frobenius error, the RMSE and the bias error are studied as the number of particle image pairs increases. The performances of 6 different models (shown in table 8) are compared on both the complete field and the field without the curvature effect. All legend items are written by combining information from columns 2 to 5 of table 8 for each row.

For every model, the Frobenius norm of the error is found to converge to a constant value after 50 epochs with an error of 0.01 (figures 34 and 35). The RMSE also converges for similar values (figures 36 and 37), while the bias error converges even faster (figures 38 and 39). Not all models converge to the same value (figures 34 and 35). The Frobenius error is visibly larger on the complete field than on the simplified field without the curvature effect. There are no significant differences between the errors on the horizontal and vertical velocity components. Figure 40 qualitatively shows the improvements of increasing the number of particle images (all predictions in this figure are obtained using the same model). It can be noted that the accuracy of predictions increases faster when the number of image pairs is smaller (look at trends for x-axis values between 1 and 10).

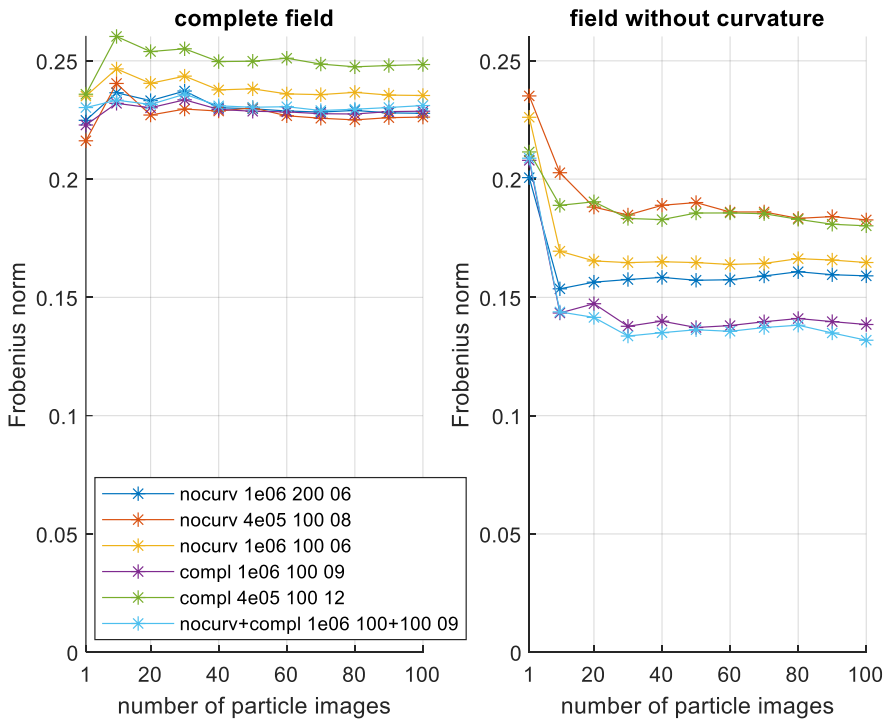

Figure 34: convergence of the Frobenius norm of the error as the number of particle images increases

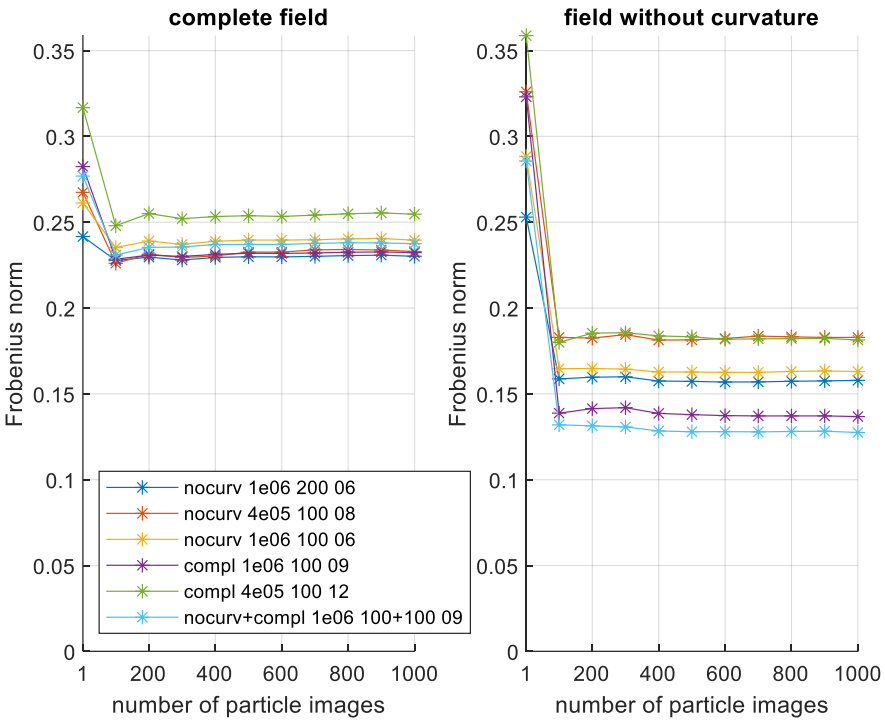

Figure 35: convergence of the Frobenius norm of the error as the number of particle images increases, zoom out

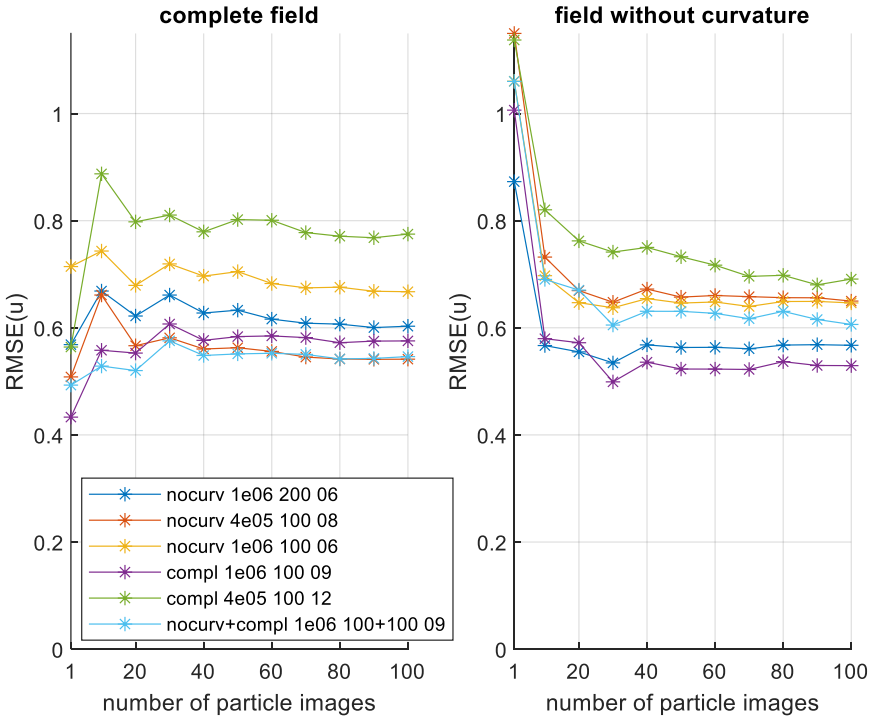

Figure 36: convergence of the RMSE on the horizontal velocity as the number of particle images increases

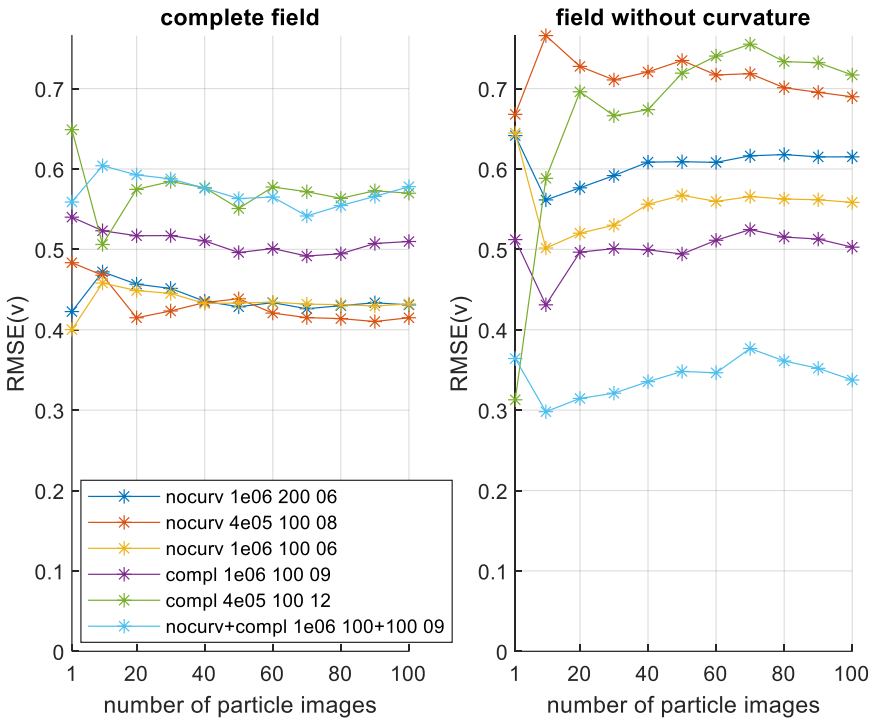

Figure 37: convergence of the RMSE on the vertical velocity as the number of particle images increases

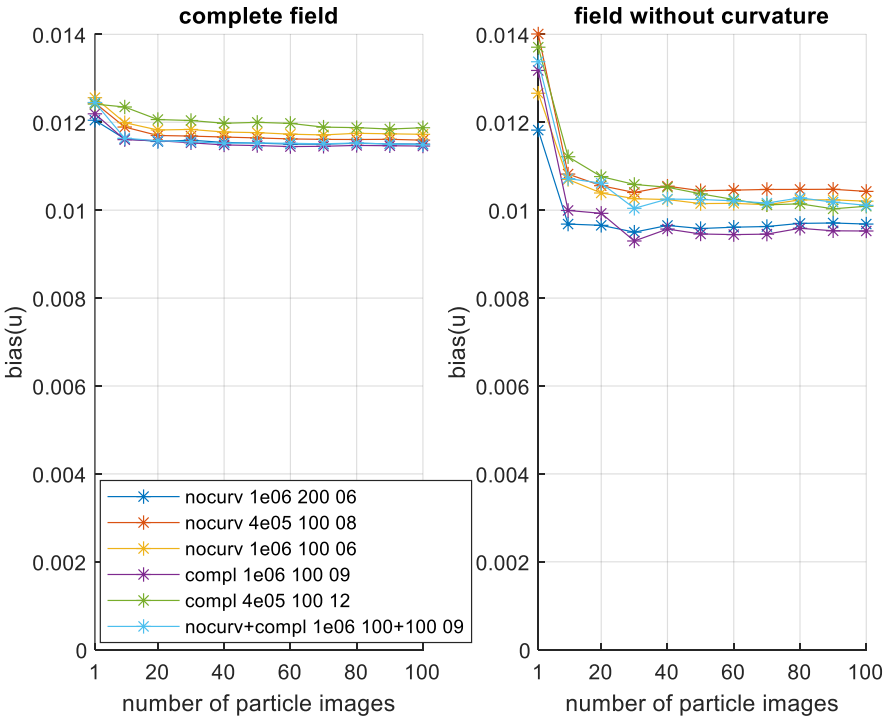

Figure 38: convergence of the bias error on the horizontal velocity as the number of particle images increases

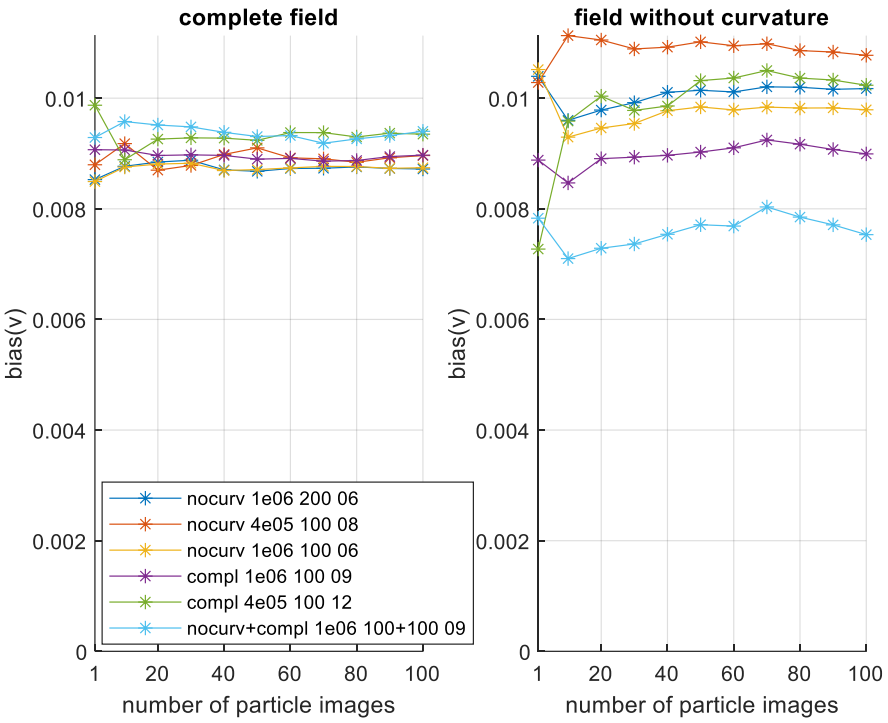

Figure 39: convergence of the bias error on the vertical velocity as the number of particle images increases

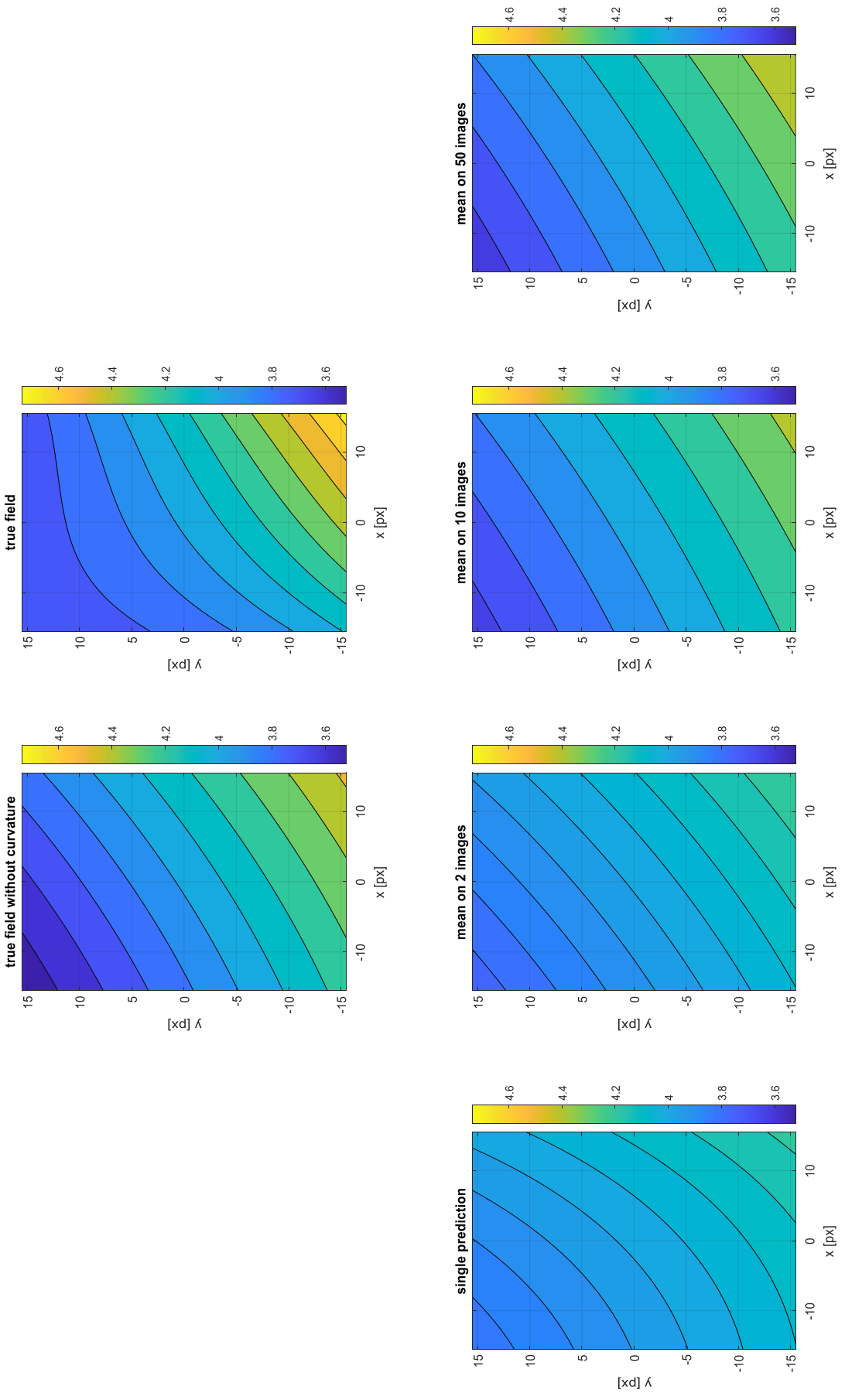

Figure 40: example of predictions as the number of image pairs increases

Each trained model of table 8 is found to react differently to fields that differ for the numeric values randomly assigned by the image generator (table 4). Therefore, the mean performances on a number of random true fields are considered to determine which model is more likely to minimize the error on random velocity fields. The convergence of the error is studied as the number of true fields increases (figure 41). 1000 different fields are generated and 50 particle image pairs are produced for each of them. We can see that the best model in figure 41 (mean performances on several random fields) differs from the best model in figure 35 (performances on one random field). The best model is chosen by considering the mean performances of the 6 trained models on both the complete fields and the fields without the curvature effect (figure 41). It corresponds to the case of training the architecture of table 5 on fields without the curvature effect for 200 epochs and a dataset of 1 000 000 data. In the following, it will be referred as Net 1. Even if the Frobenius norm does not perfectly converge on the fields without curvature (figure 41, dx), the curves do not intersect each other, so that we can reasonably assume that the best trained model would not change if the number of different fields further increased.

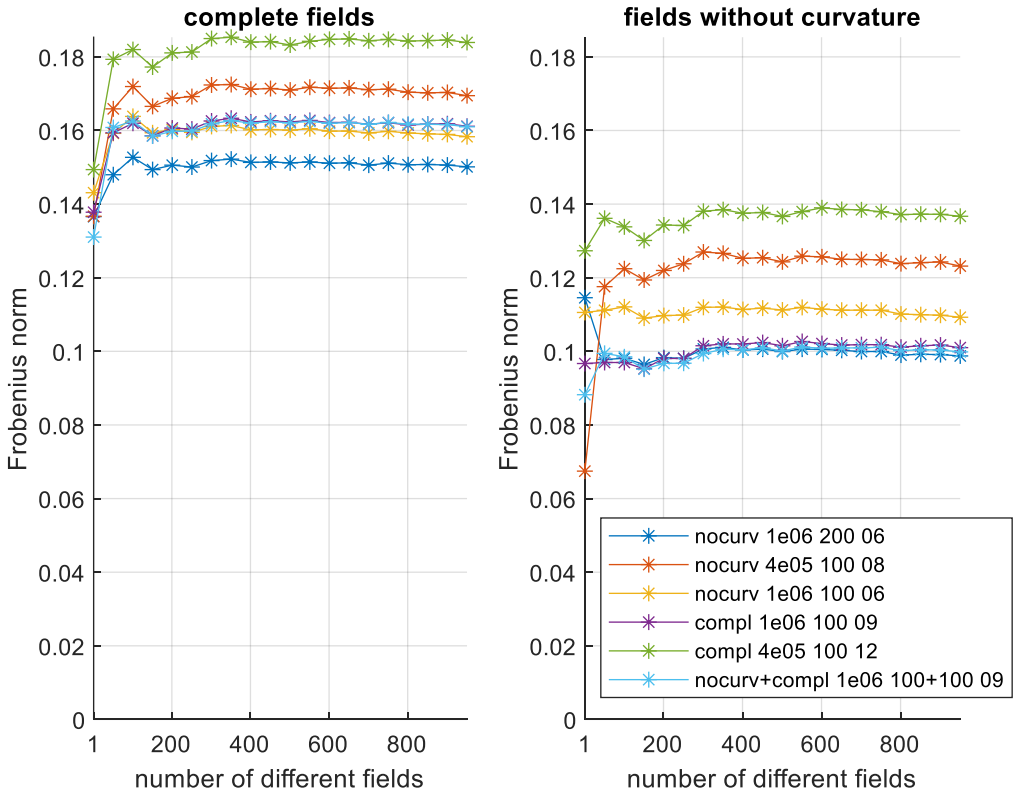

Figure 41: convergence of the Frobenius norm of the error as the number of different fields increases

4.6.2. Test 2: large synthetic particle images

Large images and benchmark velocity fields are considered. This test aims to study the effect of:

- resolution of particle images;
- computing the mean field from a number of particle image pairs;
- computing the mean field of predictions from different networks.

In the first part of test 2, Net 1 (table 8) is used to evaluate the effect of resolution on both single predictions and mean fields of 50 particle image pairs (50 comes from test 1). Three resolutions are considered: 384x192 px, 768x384 px, 1536x768 px. Both the Rankine vortex (figure 42) and the Rankine vortex with shear (figure 44) are used as benchmarks. The Rankine vortex is placed at the center of each particle image and its radius is varied proportionally with the resolution. The shear effect ($\frac{\partial u}{\partial y} \neq 0$ from section 4.5) varies proportionally to the resolution. A core radius of 75 px and the gradient of $\frac{\partial u}{\partial x} = 0.01$ are used with the smallest resolution.

The computation of the Frobenius norm of the absolute error shows that accuracy of predictions can be improved by increasing the resolution of particle images or the number of particle images related to the same velocity field or both (figure 46). The resolution and the number of particle images have the same effect on the RMSE (figure 47) and the bias error (figure 48) as on the Frobenius norm of the error. No significant differences can be observed on the horizontal and vertical velocity components. Qualitative results can be seen in figures 43 and 45. No significant differences in errors can be observed on the two different benchmarks (figure 46, 43, 45). On the other hand, a different level of accuracy is reached in different regions of the same reconstructed field, even in symmetric velocity field as the Rankine vortex is (figure 43). This effect cannot be reduced by varying either the resolution or the number of image pairs (figures 42 and 43).

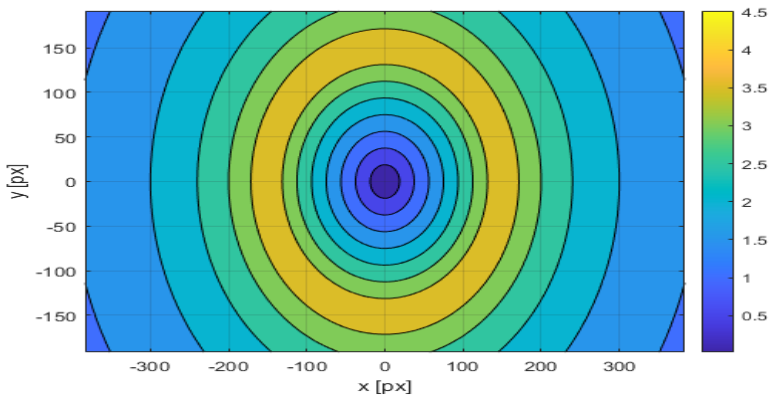

Figure 42: Rankine vortex, true field

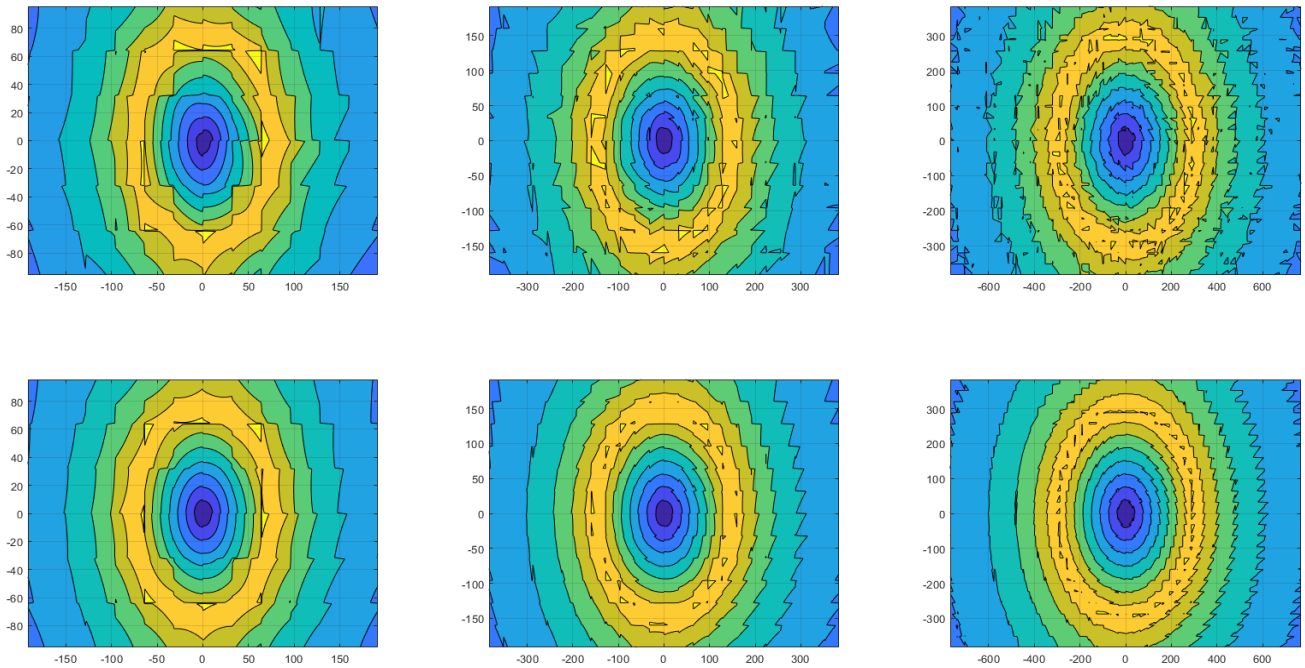

Figure 43: Rankine vortex, effect of resolution (columns: 384x192, 768x384, 1536x768) and number of image pairs (rows: 1, 50)

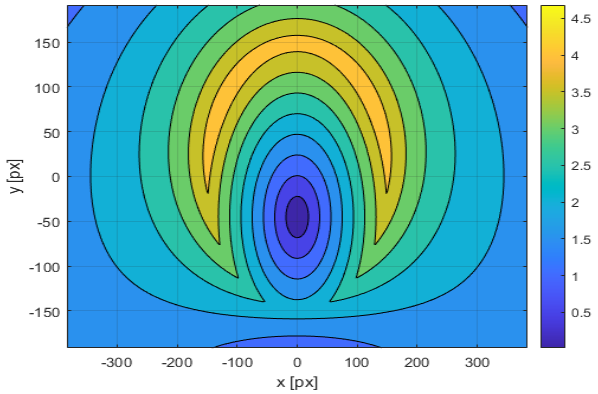

Figure 44: Rankine vortex with shear, truel field

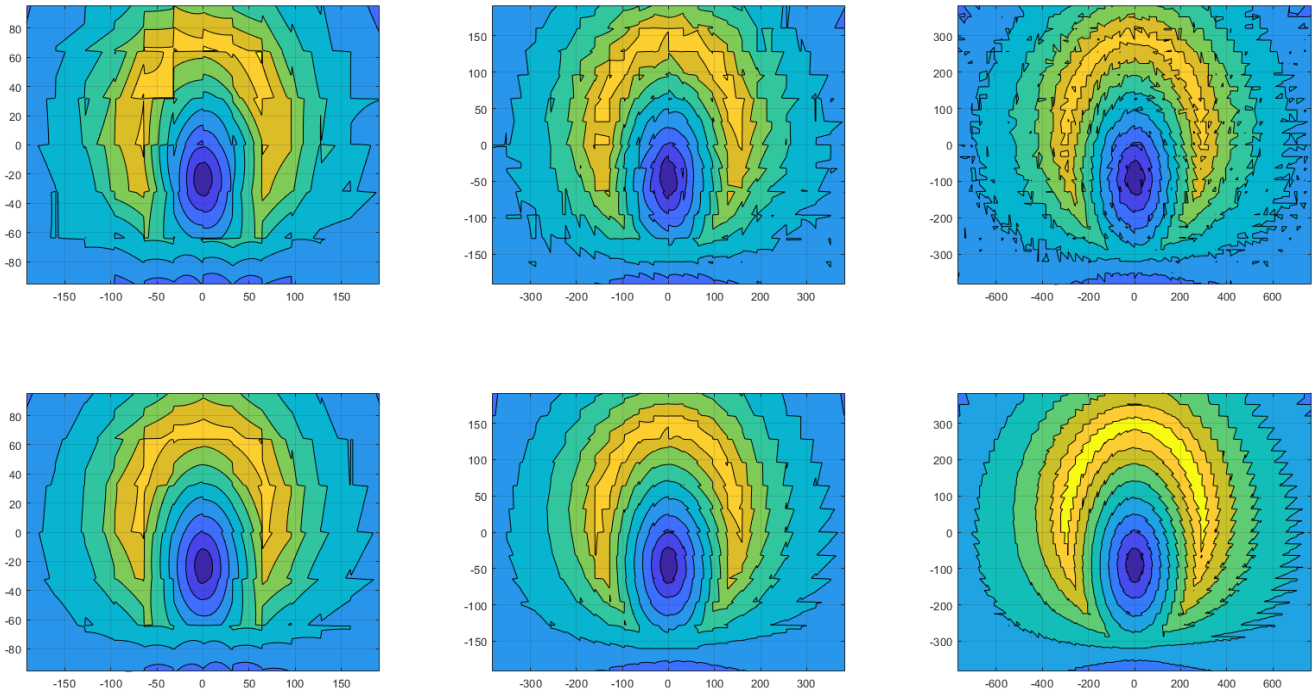

Figure 45: Rankine vortex with shear, effect of resolution (columns: 384x192, 768x384, 1536x768) and number of image pairs (rows: 1, 50)

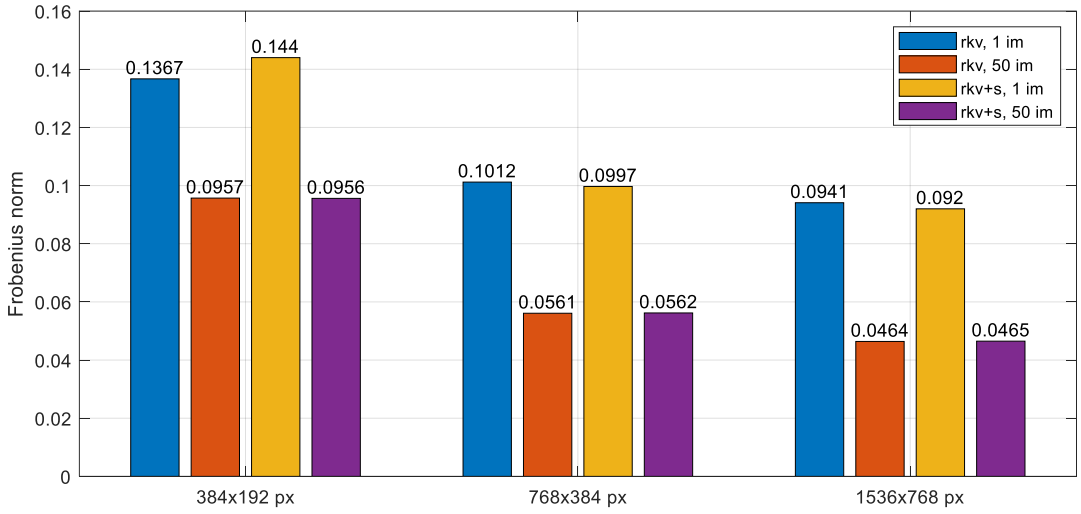

Figure 46: Frobenius norm of the error on Rankine vortex and Rankine vortex with shear, effect of resolution and number of image pairs

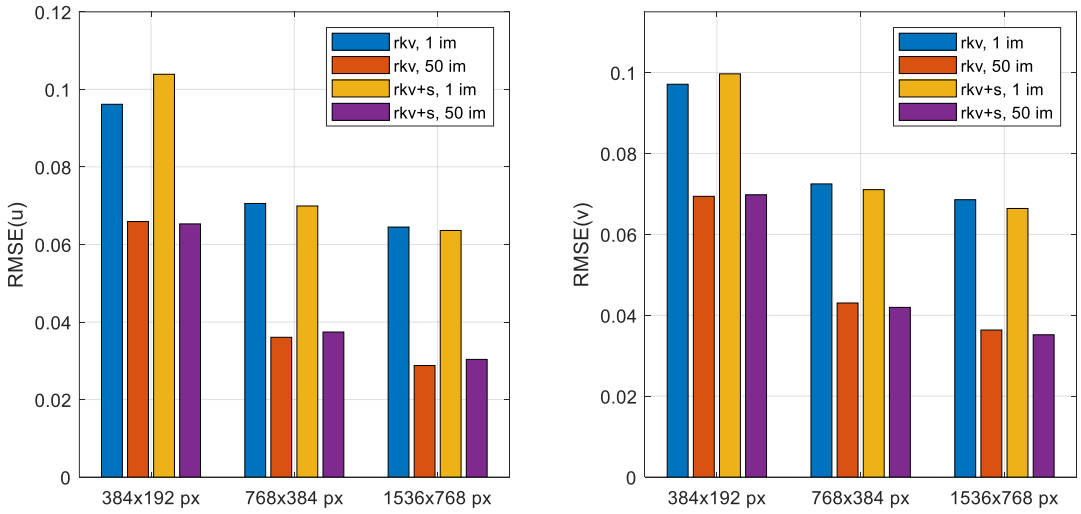

Figure 47: RMSE on Rankine vortex and Rankine vortex with shear, effect of resolution and number of image pairs

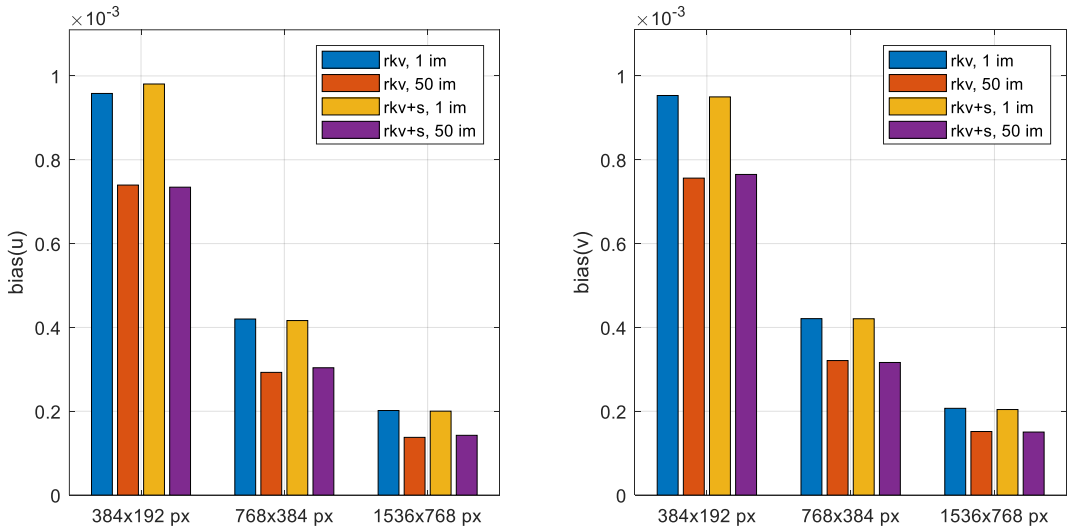

Figure 48: bias error on Rankine vortex and Rankine vortex with shear, effect of resolution and number of image pairs

In the second part of test 2, all models from table 8 are used to separately predict the velocity field, then the mean field of predictions by a group of models is computed. 50 image pairs of the same true field are used for each prediction. Rankine vortex is considered as benchmark.

In figure 49 results of predictions computed by using single models from table 8 are compared. Net 1 is confirmed to minimize the error also on benchmark fields as well as in test 1. Basically, RMSE (figure 50) and bias error (figure 51) follow the trend of the Frobenius norm of the error.

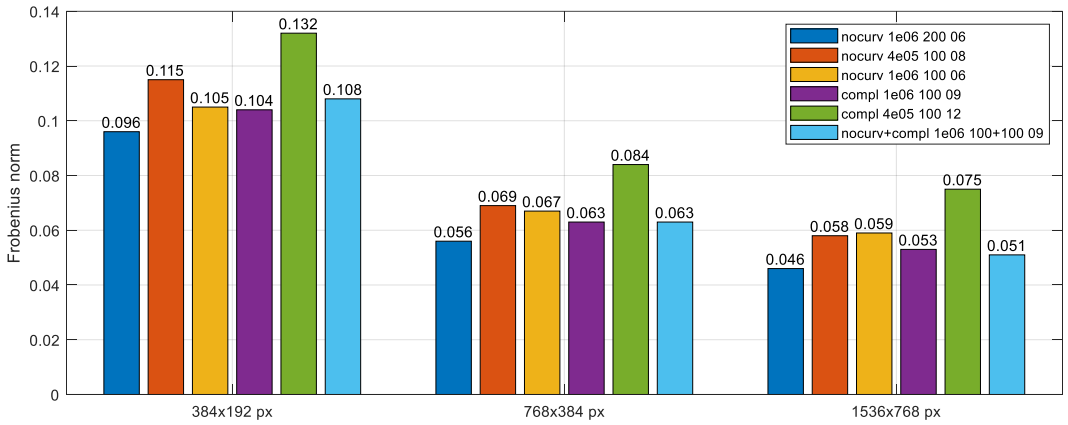

Figure 49: Frobenius norm of the error on Rankine vortex, effect of resolution and different network

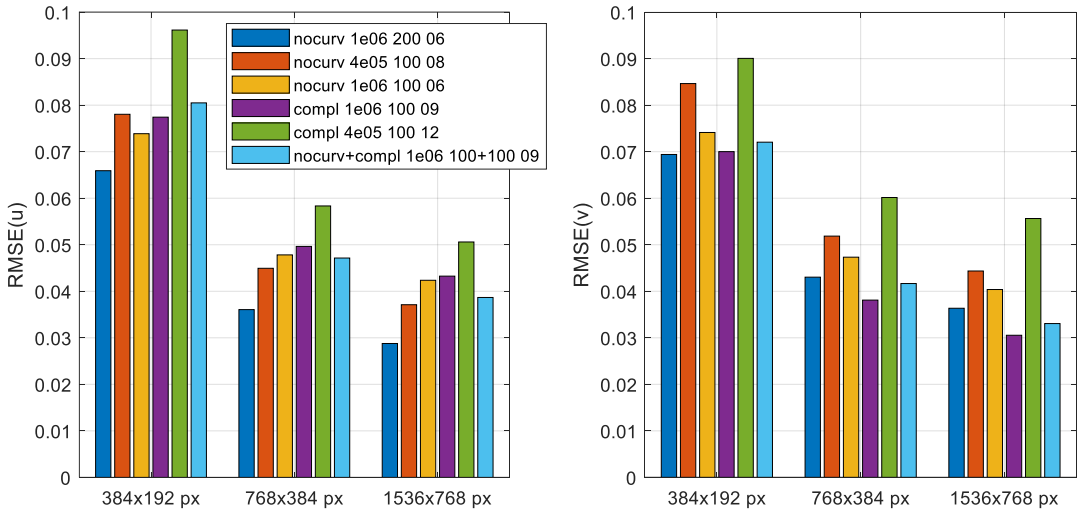

Figure 50: RMSE on Rankine vortex, effect of resolution and different networks

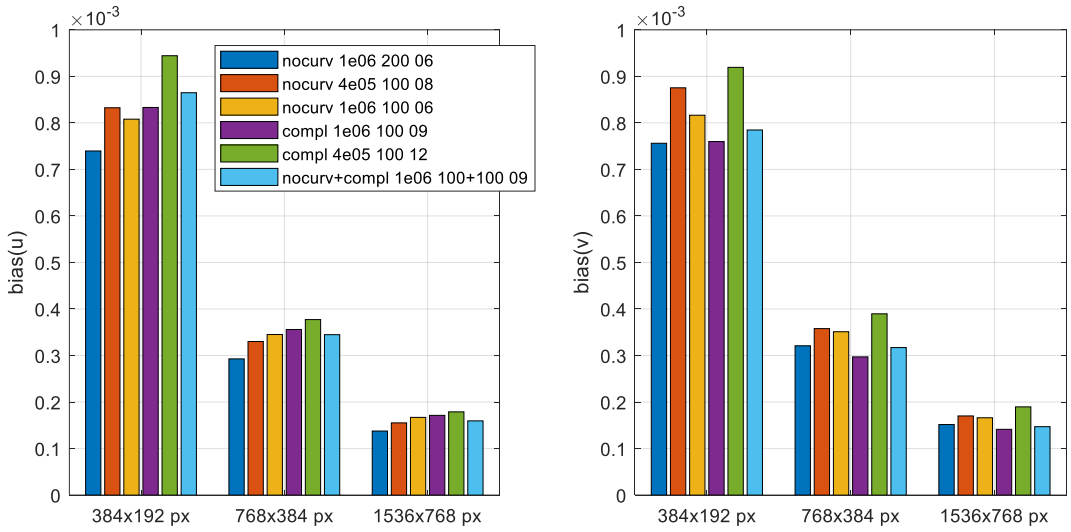

Figure 51: bias error on Rankine vortex, effect of resolution and different networks

The prediction computed by the best single model (net 1, figure 49) is compared to the mean prediction of different groups of models using the Frobenius norm of the error, the RMSE and the bias error (figures 52, 53, 54). This kind of mean prediction is found to further reduce the Frobenius norm of the error with reference to figure 49, especially with higher resolutions (figure 52). The best accuracy prediction is reached by combining results from 5 models of table 8 (figure 55) and even the model that singly produce the higher error (net 5 from table 8) is useful to reduce the error of the mean prediction. This means that each model tends to wrong in a different way.

The effect of using a different number of models to compute predictions is qualitatively shown in figures 56, 57 and 58 where the best prediction from a single model (figure 56) is compared to the best prediction from a couple of models (figure 57) and to the best prediction from all models from table 8 (figure 58).

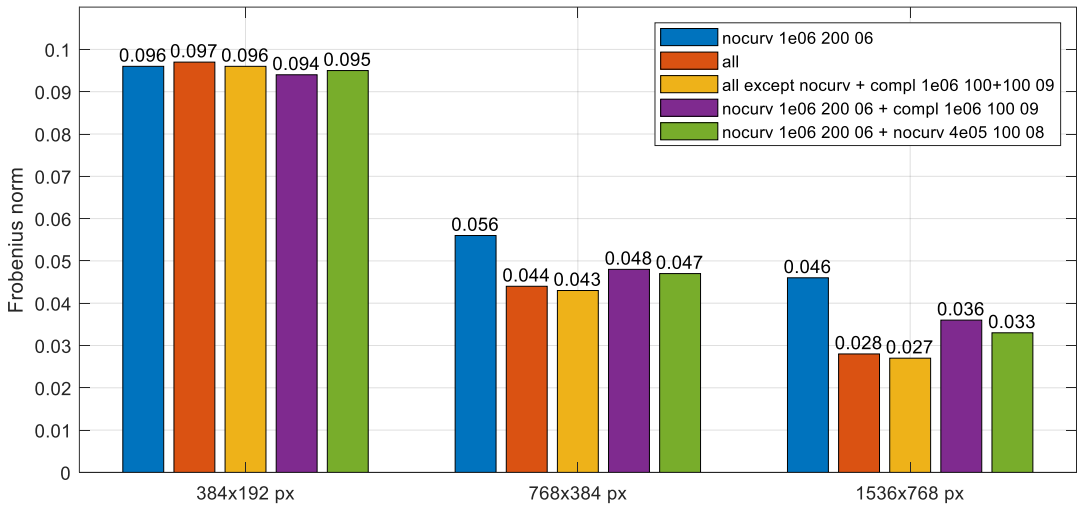

Figure 52: Frobenius norm of the error on Rankine vortex, effect of resolution and mean predictions using different groups of networks

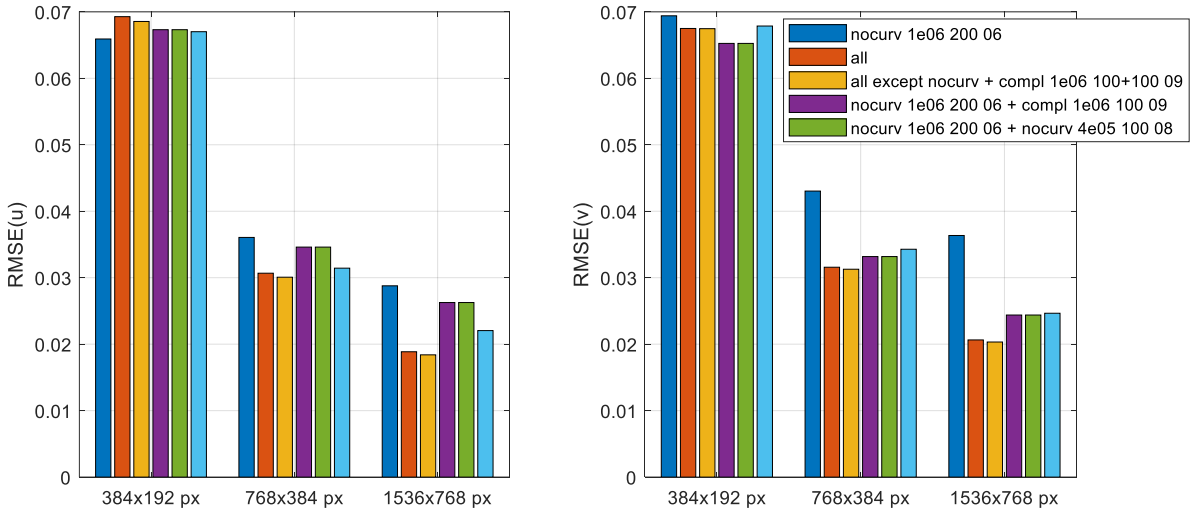

Figure 53: RMSE on Rankine vortex, effect of resolution and mean predictions using different groups of networks

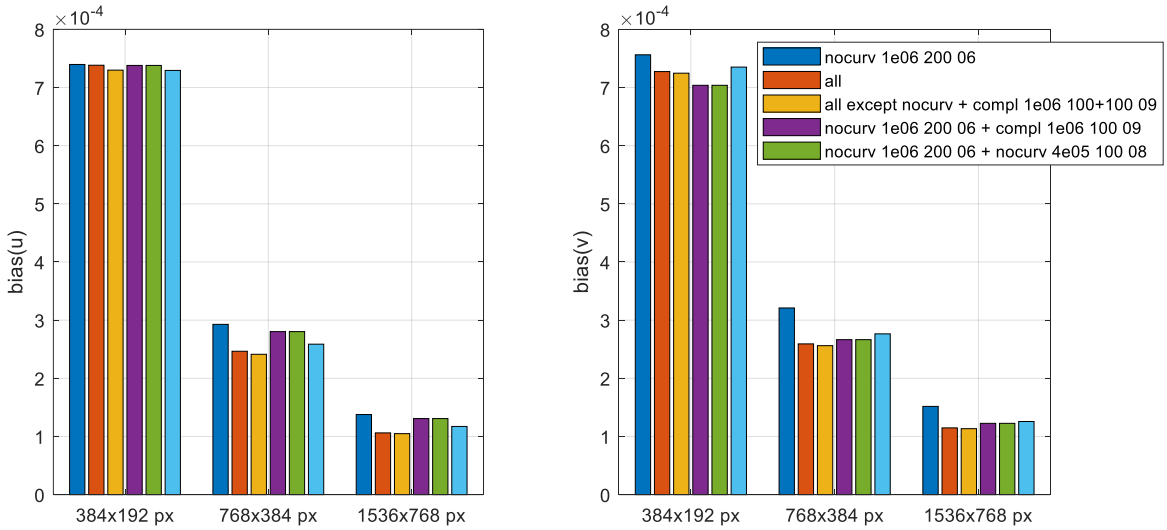

Figure 54: bias error on Rankine vortex, effect of resolution and mean predictions using different groups of networks

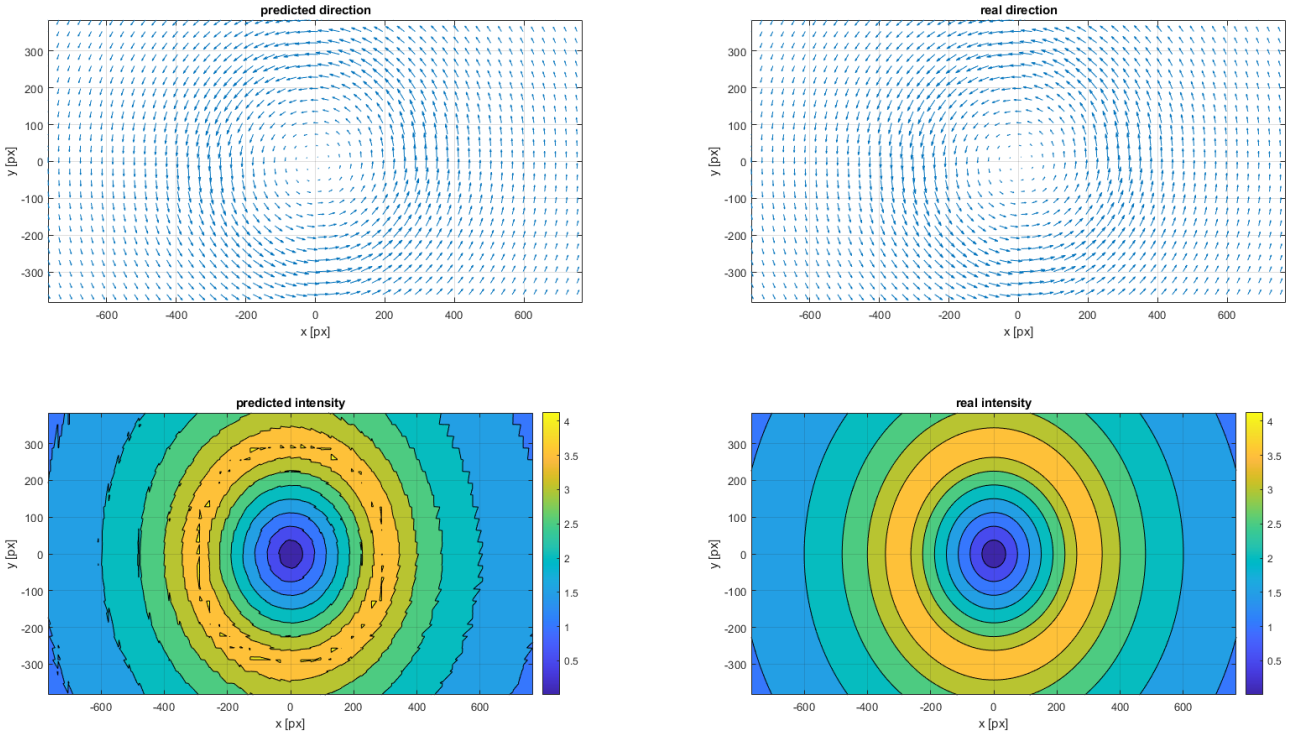

Figure 55: best accuracy attempt, direction and intensity

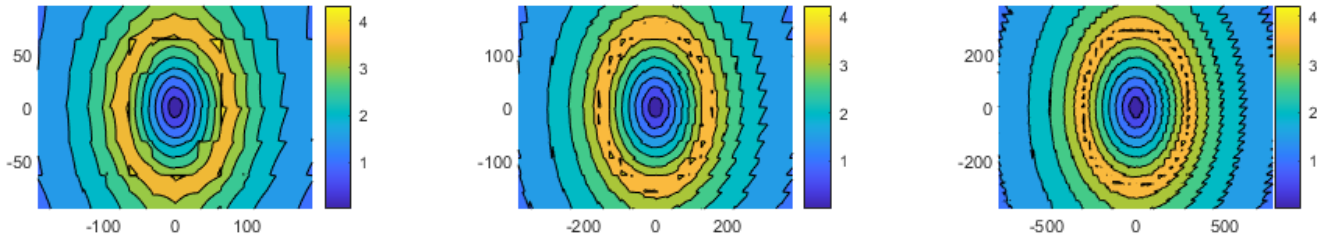

Figure 56: best prediction by using a single network ('nocurv 1e06 200 06'), effect of resolution

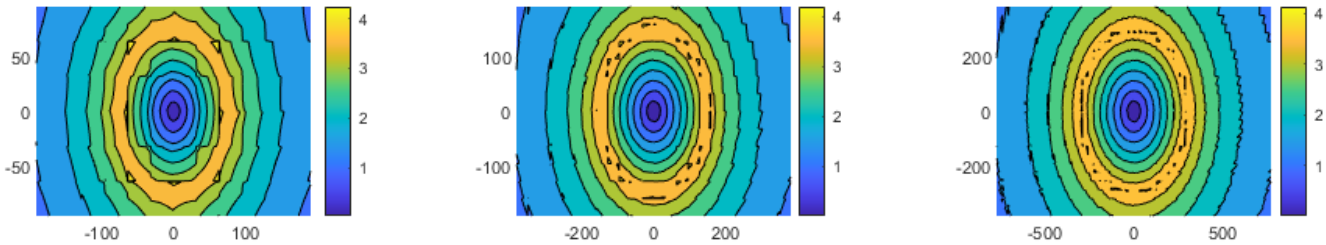

Figure 57: best accuracy attempt (mean prediction from all networks except 'nocurv+compl 1e06 100+100 09'), effect of resolution

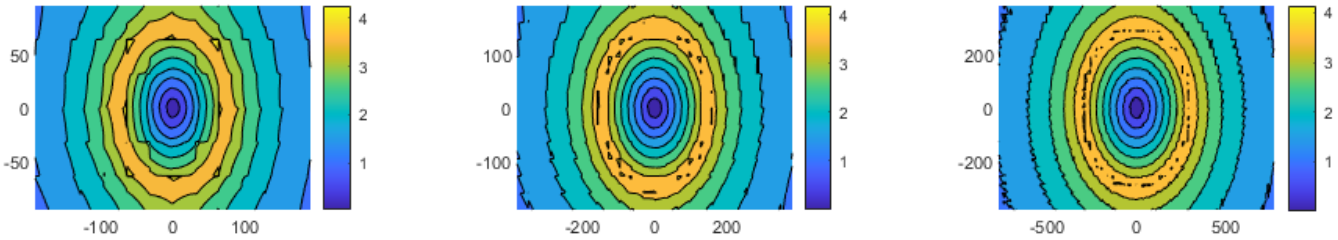

Figure 58: best accuracy attempt from a couple of models ('nocurv 1e06 200 06' and 'nocurv 4e05 100 08'), effect of resolution

4.6.3. Test 3: variations in particle density and displacement range

This test aims to determine if the particle image density affects performances of the trained models, since it is fixed at 0.029 ppp for the whole training process (section 4.2). Synthetic images characterized by different particle densities are taken from the First International PIV Challenge [48]. All the images are related to the same velocity field and they differ for the following features:

- particle density can be high, medium or low;
- particle size can be small or large.

The most accurate model on both test 1 and test 2 (net 1 of table 8) is selected for the test.

First, the images are analyzed using PIVlab (figure 59), an open-source tool for performing PIV analysis in Matlab [49]. PIVlab reconstructions are based on the cross-correlation method and are here used as a reference for the evaluation of deep learning performances. From PIVlab analysis (figure 59), the velocity magnitude is found to exceed the training displacement range in some regions of the images (displacement range is estimated to be about [-10, 10] px/frame, while the training displacement range is [-4, 4] px/frame).

Results are shown in figures from 60 to 65. Performances are found to be essentially independent from particle density and size. In the case of low particle density (figures 64 and 65), the intensity predictions may seem to be less accurate than in the case of medium and high particle density, but this effect can be attributed to the effective lack of particles in certain zones of the input images. In all cases, the training displacement range is found to limit the predictions of the absolute velocity intensity to values within it, although it allows to reconstruct velocity direction and relative velocity intensity.

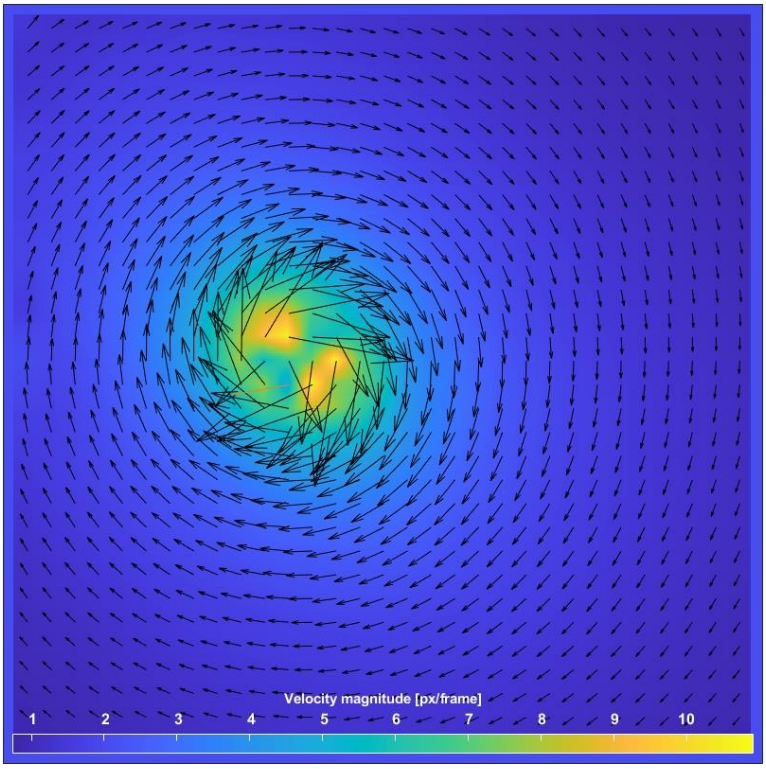

Figure 59: reconstruction by PIVlab, reference for test on variable particle density

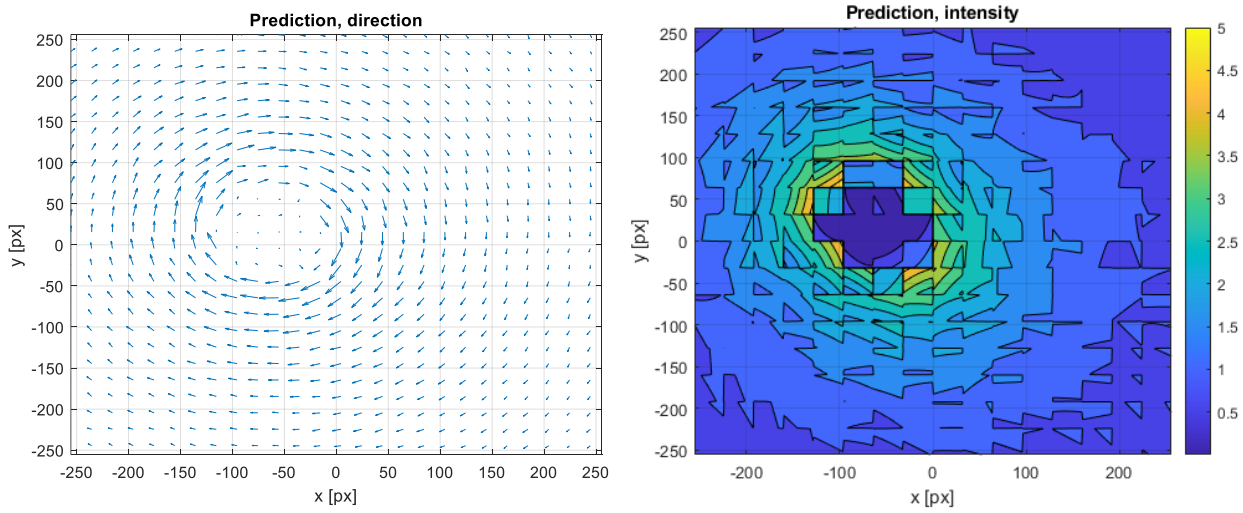

Figure 60: deep learning with high particle density and small particle, synthetic images

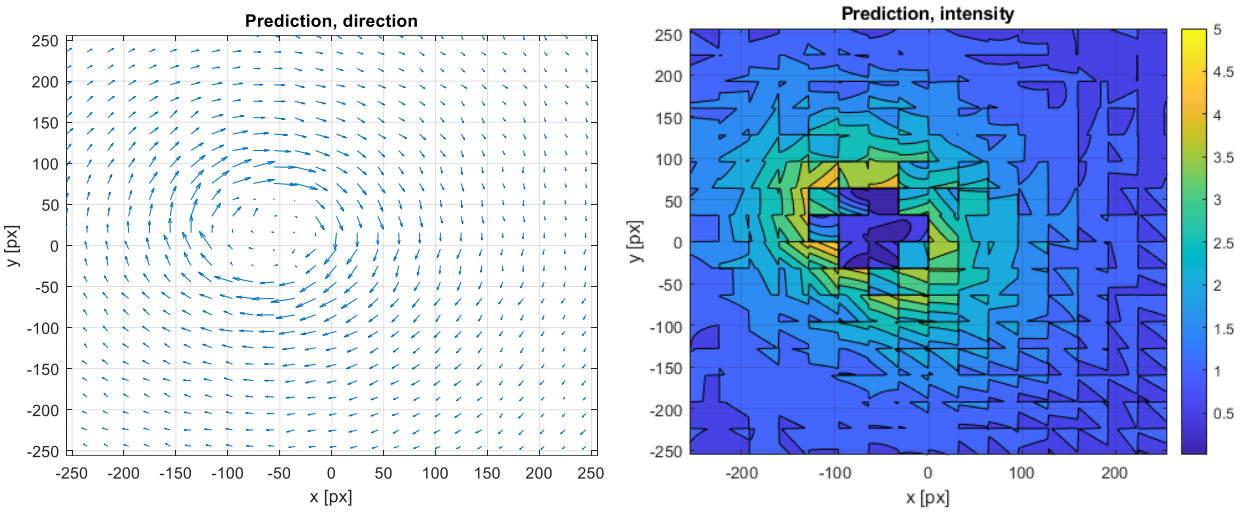

Figure 61: deep learning with high particle density and large particle, synthetic images

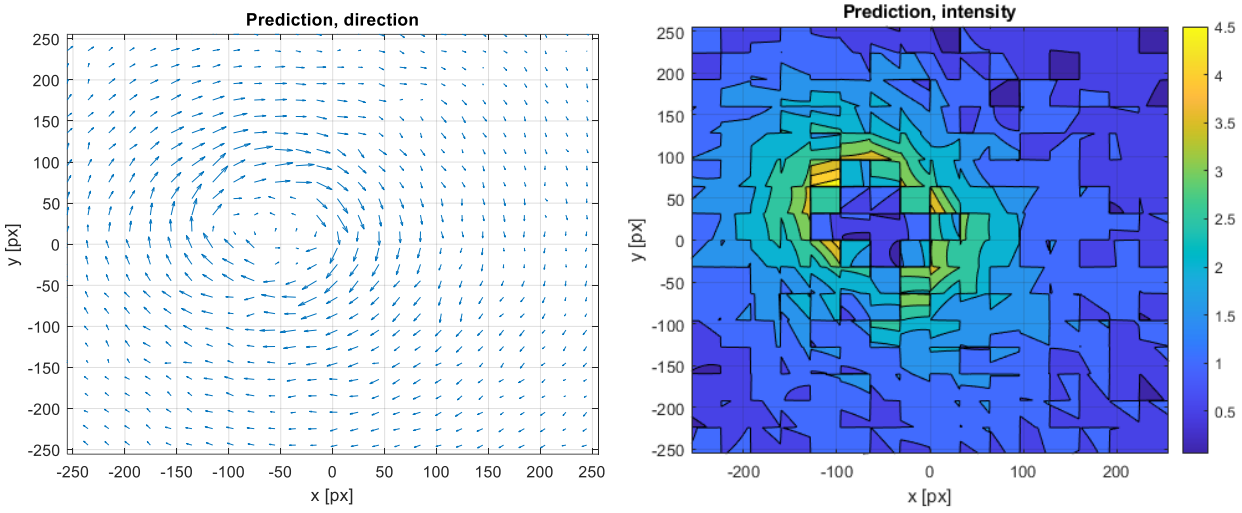

Figure 62: deep learning with medium particle density and small particle, synthetic images

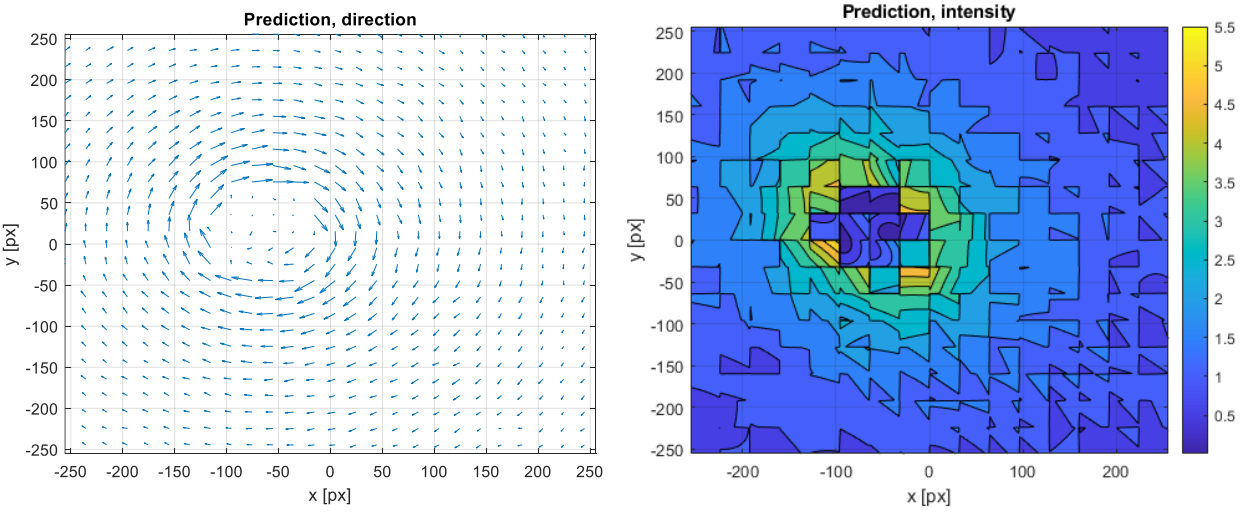

Figure 63: deep learning with medium particle density and large particle, synthetic images

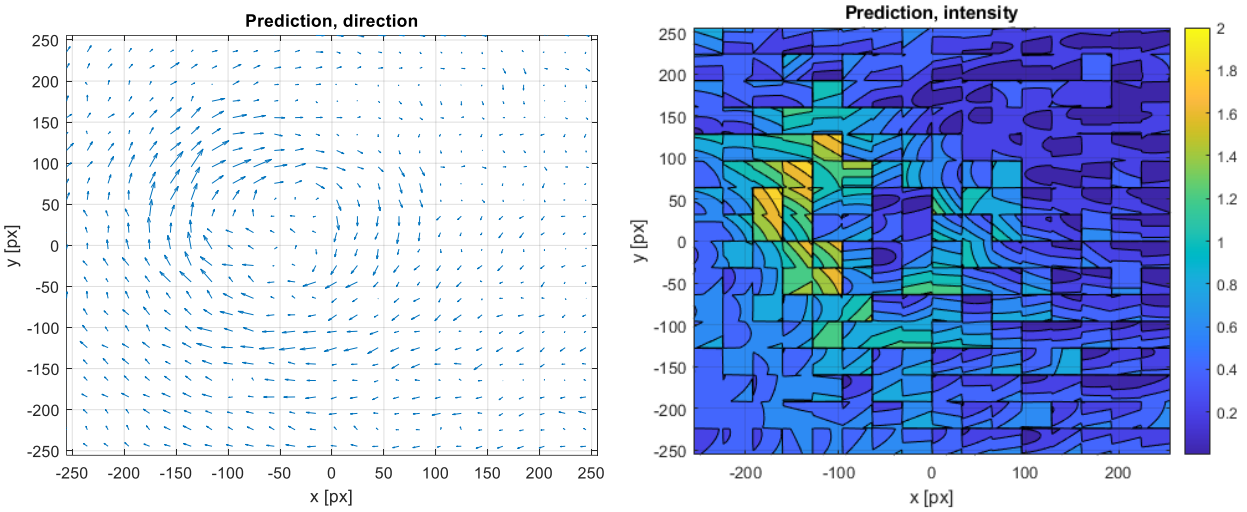

Figure 64: deep learning with low particle density and small particle, synthetic images

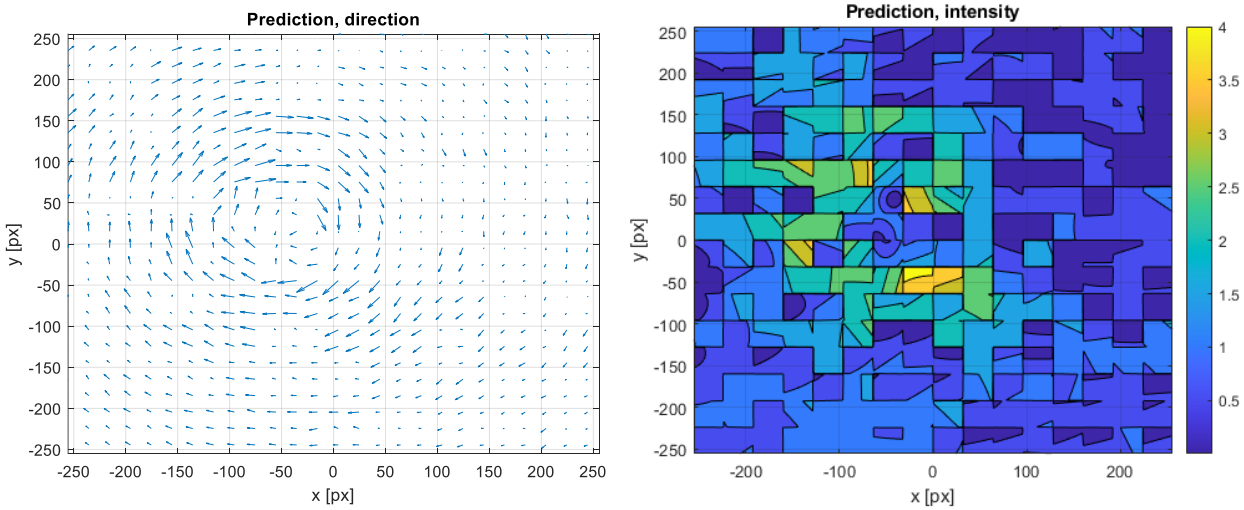

Figure 65: deep learning with low particle density and large particle, synthetic images

4.6.4. Test 4: real particle images

This test aims to evaluate CNN performances on real PIV images. Two real experiments are considered: a turbulent jet and a turbulent boundary layer. As in test 3, PIVlab is used to provide reference fields (figures 66 and 67).

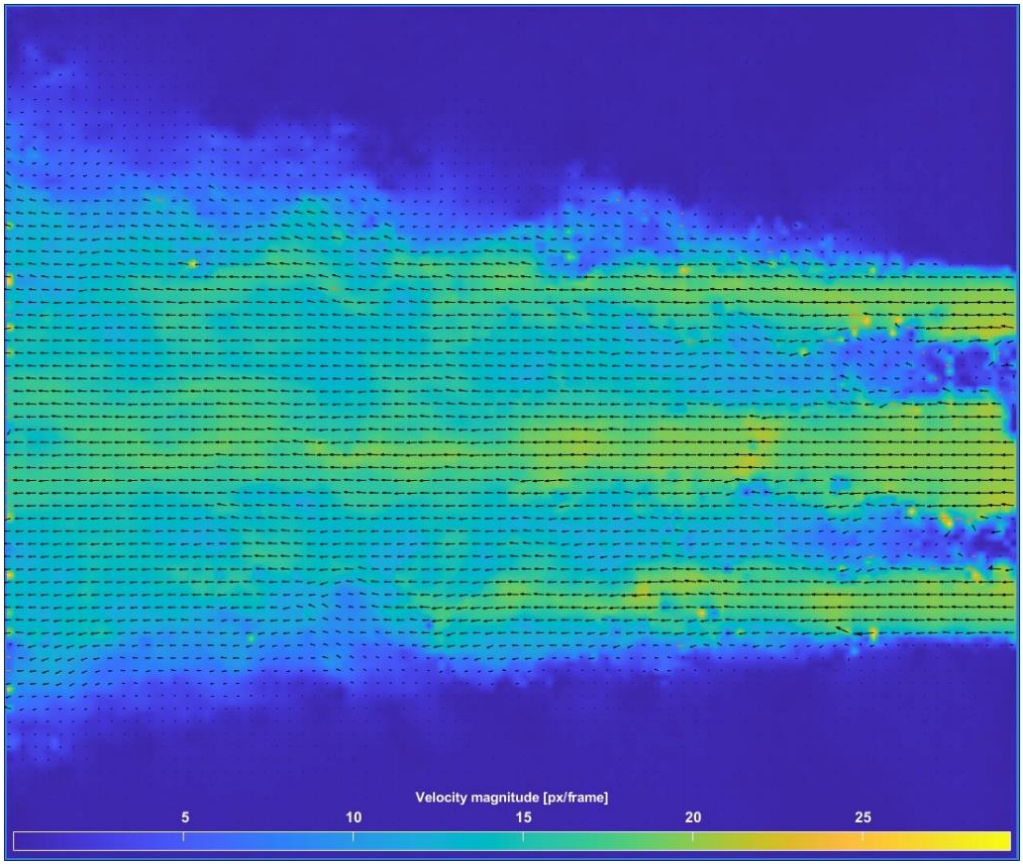

Figure 66: reconstruction of a real turbulent jet by PIVlab

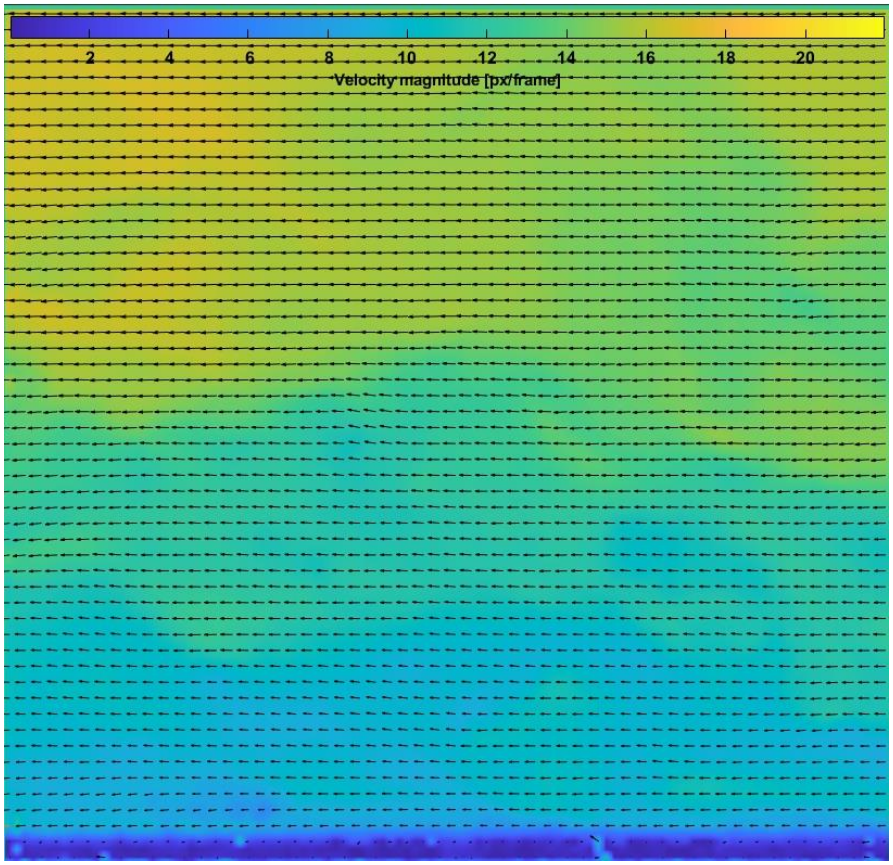

Figure 67: reconstruction of a real turbulent boundary layer by PIVlab

No good performances are achieved using CNN models (table 9) on both the real jet and the real boundary layer (figures 70 and 72). The reference study by Rabault et al. achieved quite good performances on real data, but the network was tested only on a low velocity water flow so that the real velocity range was equal to the training velocity range (from -4 px/frame to 4 px/frame) [15].

In general, the main difference between synthetic and real data is the amount of noise: the real images are noisier than the synthetic ones. In the present work, it is logic to suppose that problems with real images are related to noise since performances on synthetic images are quite good.

The following comparisons are implemented to verify this hypothesis and to detect other possible causes of poor performances:

- The same images are processed using two networks that differ only for the noise used in the training process: Net 1 of table 9 uses white noise with zero mean and variance 1% of the maximum pixel value of the image in the whole training process (the same noise is used by Rabault et al. [15]), while Net 7 of table 9 is trained on images without noise. Qualitative results are shown in figures 68 and 69. It can be noted that predictions of the two networks are different, although they are both poor (particularly, the net trained without noise seems more sensitive to small pixel intensity variations, see figure 69). This means that the noise of the synthetic training dataset is somehow incorporated by the network during training. However, the amount of training noise seems insufficient. In future works, the effect of white noise with zero mean and variable variance in the training process can be studied. If performances are still unsatisfactory, other kinds of noise can be added to the training images.

- The same images are processed after the application or not of different filters in a preprocessing step in PIVlab. Contrast-limited adaptive histogram equalization (CLAHE) and intensity capping filter (CLAHE locally enhances the contrast in the images, intensity capping selects an upper limit of the greyscale intensity and replaces all pixels that exceeds the threshold by this upper limit) seem to slightly improve performances. Qualitative results are shown in figures from 70 to 73. Wiener2 denoise filter, high-pass filter and contrast stretch are tried unsuccessfully. Since the main feature of PIVlab is to be user-friendly, it allows to apply preprocessing filters in a simplified way. For example, it is not possible to apply the same filter at different noise levels in PIVlab. In future works, a more advanced software than PIVlab can be used to further investigate effects of preprocessing by filtering noise at different levels using the same kind of filter. However, even the simplified analysis here presented allows to confirm that poor performances of trained networks are related to noise sensitivity.

- The same images are processed using two networks that differ only for the displacement range of the training velocity fields: Net 1 (table 9, this is the model that gives best performances on tests 1 and 2) is trained on [-4, 4] px/frame while Net 8 (table 9) is trained on [-10, 10] px/frame. Qualitative results are shown in figures 68 and 70. Predictions are confirmed to be limited by the training displacement range, as shown in test 3. It is fundamental to overcome this limitation to use neural networks in real PIV experiments. Theoretically, it can be easily done by include higher displacements in the training dataset.

Net	Training fields	Training data	Training displacement range [px/frame]	Epochs	Noise
1	without curvature	1 000 000	-4 ÷ 4	200	white noise (section 4.2)
7	without curvature	1 000 000	-4 ÷ 4	200	no noise
8	without curvature	1 000 000	-10 ÷ 10	200	white noise (section 4.2)

Table 9: networks for test on real images

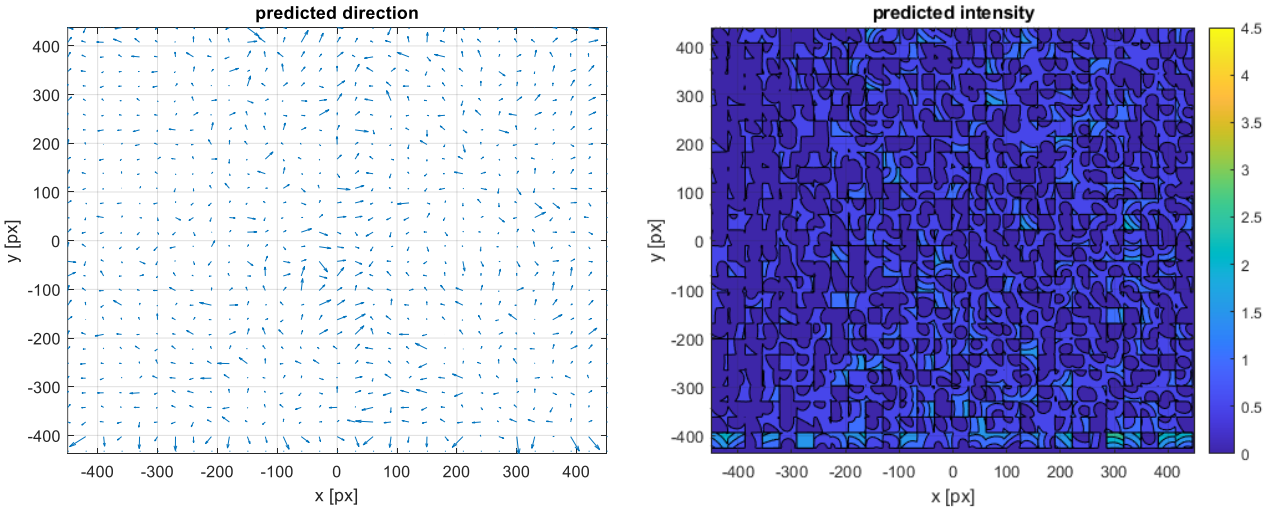

Figure 68: prediction from Net 1 (noisy training images and [-4, 4] px/frame training displacements, see table 9) without preprocessing, turbulent boundary layer

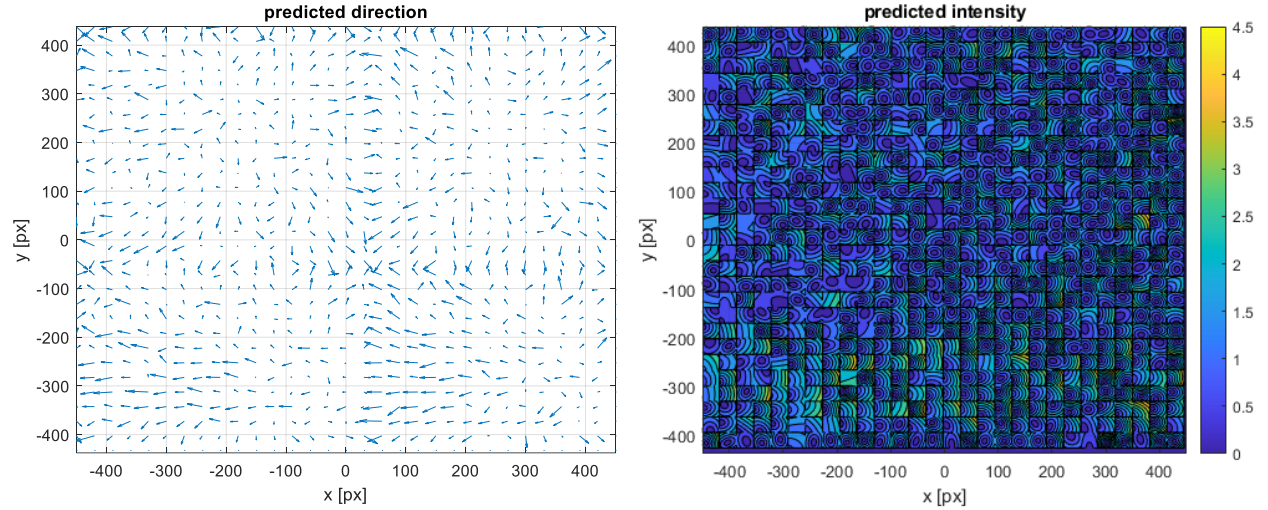

Figure 69: prediction from Net 7 (perfect training images and [-4, 4] px/frame training displacements, see table 9) without preprocessing, turbulent boundary layer

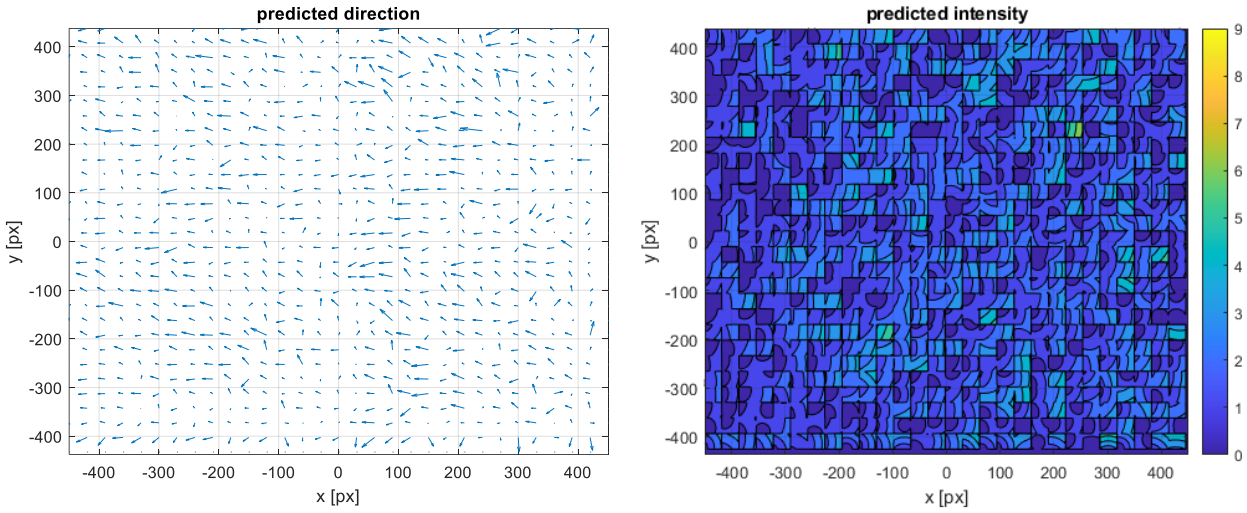

Figure 70: prediction from Net 8 (noisy training images and [-10, 10] px/frame training displacements, see table 9), without preprocessing, turbulent boundary layer

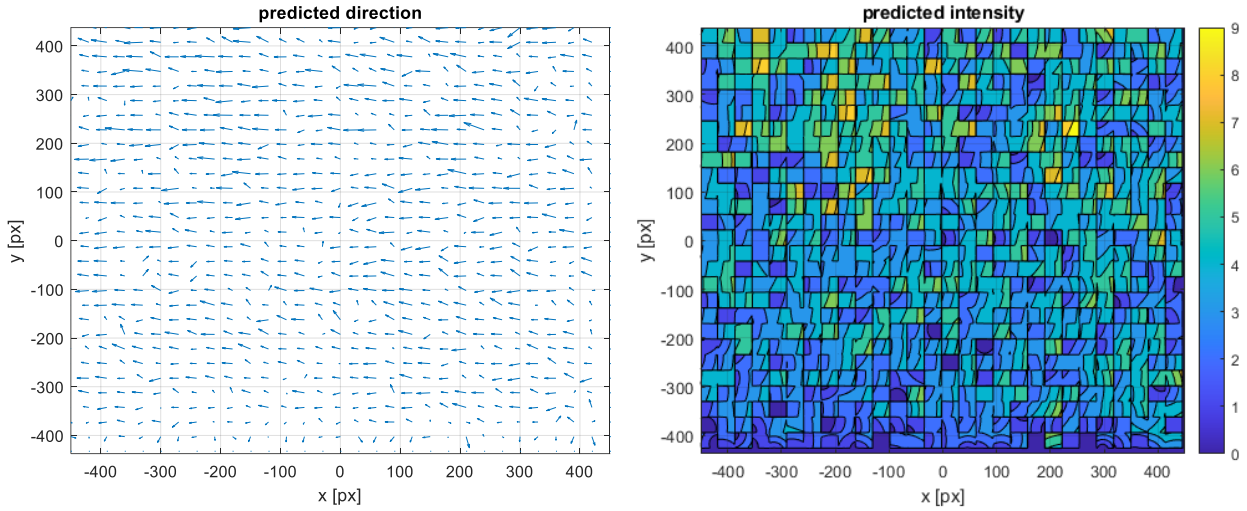

Figure 71: prediction from Net 8 (noisy training images and [-10, 10] px/frame training displacements, see table 9) after preprocessing with CLAHE and intensity capping, turbulent boundary layer

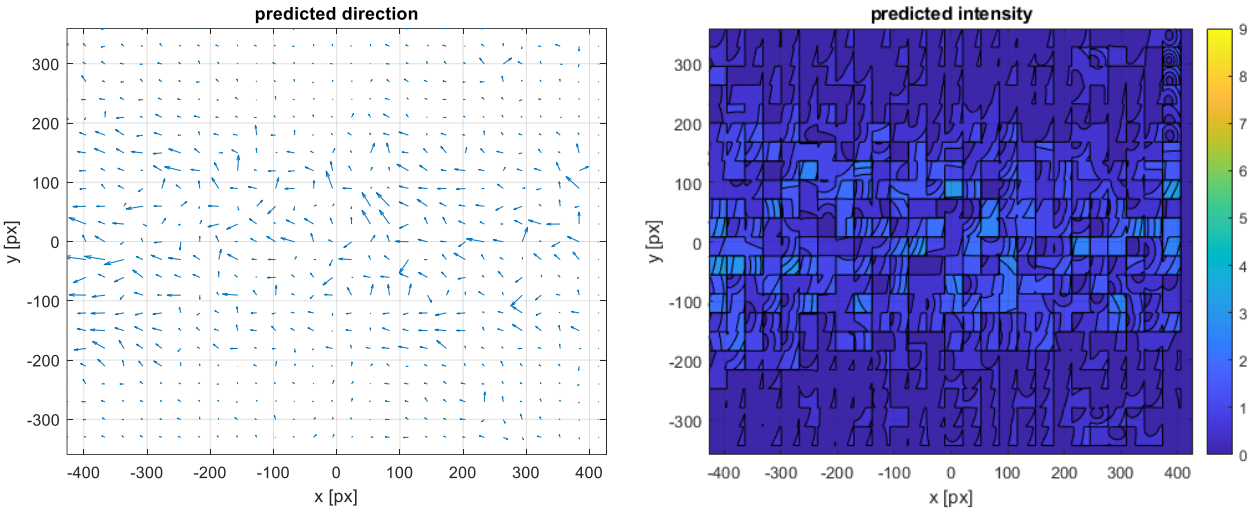

Figure 72: prediction from Net 8 (noisy training images and [-10, 10] px/frame training displacements, see table 9) without preprocessing, turbulent jet

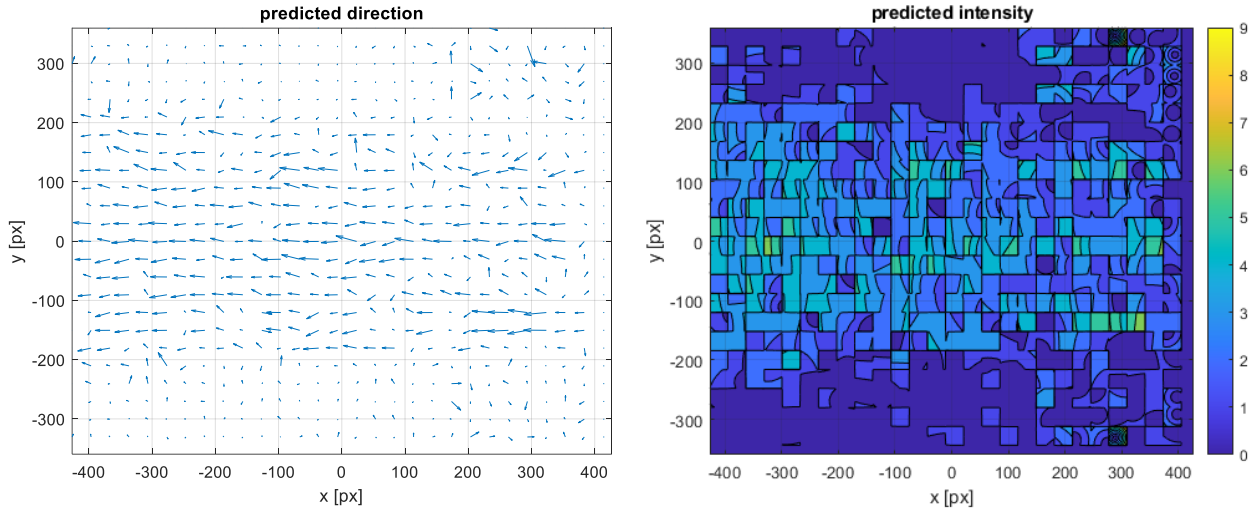

Figure 73: prediction from Net 8 (noisy training images and [-10, 10] px/frame training displacements, see table 9) after preprocessing with CLAHE and intensity capping, turbulent jet

5. Conclusions and future works

The present work aims to perform standard PIV using CNNs. Two kinds of training are applied. They differ for the mathematical structure of the velocity field used to move seeding particles from the first to the second frame. In the first case, velocity fields are represented by first order polynomials, in the second case by second order polynomials. In both cases the network is trained to predict first order approximations of the velocity fields. Best results are achieved by applying the first kind of training that allows to reduce the final training RMSE significantly more than the second one, though the second kind is found to work better than the first if the same final training RMSE is achieved by the two trained models and even if the second model has a slightly higher final RMSE than the first. Probably, the second training produces networks more robust to noise by training them to distinguish gradient effect from curvature.

The best single network prediction achieves the value of 0.46 for the Frobenius norm of the error on both the benchmarks using 50 image pairs of size of 1536x768 pixels at least. Accuracy is expected to decrease as the resolution further increases, since the network is trained to predict a first order estimation of the velocity field and the real field becomes more similar to the first order approximation as the ratio between the size of the interrogation window and the whole image decreases. The reconstruction of the velocity field depends on the random distribution of seeding particles in the input images. This effect can be reduced by computing the mean field from several distributions of particles, despite increasing the computational cost. The accuracy is verified to not decrease significantly by computing the mean field for more than 50 image pairs.

The Frobenius norm of the error can be further reduced by computing the mean field of predictions from different models. The value of 0.27 is achieved for the Frobenius norm of the error by using 5 different networks, despite increasing the computational cost. A value of 0.33 is achieved by averaging results from 2 models that only differ for the size of the training dataset. No significant differences are observed on predictions on the two benchmarks: the model seems to be able to generalize.

Performances are found to be essentially independent from particle density and size. Conversely, the training displacement range is found to limit the predictions of the absolute velocity intensity to values within it, although it allows to reconstruct velocity direction and relative velocity intensity. It seems that training on larger displacements can be performed without theoretical difficulties by increasing the size of the training dataset.

No good performances are achieved on real images. This is essentially attributed to the noise of real data that is higher than that used during training on synthetic images. The application of CLAHE and intensity capping seems to slightly improve performances.

Matlab 2019b is found to be suitable to train complex deep learning projects, differently from previous releases. It seems to be a credible alternative to open-source libraries for people that are already comfortable with Matlab. Although TensorFlow is still probably better to push the boundaries of deep learning, Matlab can be considered to solve simple or even complex problems with established deep learning techniques.

Neural networks can probably become a credible alternative to cross correlation to perform PIV. However, further investigations are needed. The present work has a number of open points that can be investigated in future works:

- training on high velocity range and high noise images should be performed to effectively apply CNNs in real applications;
- the network architecture can be modified to predict also the curvature effect on each interrogation window;

- the accuracy of the single model can be improved through further training and hyperparameter tuning;
- code efficiency can be improved to speed up the prediction process (parallel computing can be considered);
- the network architecture and the generation of synthetic data can be modified to apply deep learning to stereoscopic and tomographic PIV.

Acknowledgements

This work was possible thank to my supervisors, dr. Francesco Della Santa and HPC@Polito.

I wish to thank prof. Sandra Pieraccini for guiding me through the whole thesis with very deep practical sense and for providing me technical support quickly every time I asked for it and sometimes even when I only had in mind to ask. I wish to thank prof. Gaetano Iuso and dr. Gioacchino Cafiero for introducing me to PIV and for enlightening me about the interpretation of results. They also provided real PIV images for test 4. I wish to thank dr. Francesco Della Santa for decisively helping me with error detection and image processing in the first phases of the training process.

I wish to thank HPC@Polito for providing students with access to a cluster with 8 Tesla GPU.

References

[1] Rosenblatt F., Principles of neurodynamics: perceptrons and the theory of brain mechanisms, Spartan Books, 1962

[2] McCulloch W. S., Pitts W., A logical calculus of the ideas immanent in nervous activity, Bull Math Biophys 5: 115-133, 1943

[3] Rumelhart D., Hinton G., Williams R., Learning representations by back-propagating errors, Nature 323: 533-536, 1986

[4] Hinton G. E., Learning multiple layers of representation, Trends Cogn Sci 11: 428-434, 2007

[5] Nielsen M., Neural networks and deep learning, Determination Press, 2015

[6] Russakovsky O., Deng J., Su H., Krause J., Satheesh S., Ma S., Huang Z., Karpathy A., Khosla A., Bernstein M., Berg A., Fei-Fei L., ImageNet Large Scale Visual Recognition Challenge, Int J Comput Vis 115: 211-252, 2015

[7] Krizhevsky A., Sutskever I., Hinton G. E., ImageNet classification with deep convolutional neural networks, NIPS 25: 1097-1105, 2012

[8] Jackson A., Bulat A., Argyriou V., G. Tzimiropoulos, Large pose 3D face reconstruction from a single image via direct volumetric CNN regression, ICCV, 2017

[9] Miao S., Wang Z. J., Liao R., Real-time 2D/3D registration via CNN regression, ISBI: 1430-1434, 2015

[10] Pyoa J., Duan H., Baek S., Kim M. S., Jeon T., Kwon J. S., Lee H., Choa K. H., A convolutional neural network regression for quantifying cyanobacteria using hyperspectral imagery, Remote Sens Environ 233: 111350, 2019

[11] Lathuilière S., Mesejo P., Alameda-Pineda X., Horaud R., A comprehensive analysis of deep regression, IEEE Trans Pattern Anal Mach Intell 41: 1-17, 2019

[12] Raffel M., Willert C., Wereley S., Kompenhas J., Particle image velocimetry: a practical guide, Springer, 2007

[13] Bastiaans R. J. M., Cross-correlation PIV: theory, implementation and accuracy, EUT reports 99-W-001, Eindhoven: Technische Universiteit Eindhoven, 1993

[14] Adrian R., Westerweed J., Particle image velocimetry, Cambridge University Press, 2011

[15] Rabault J., Kolaas J., ensen A., Performing particle image velocimetry using artificial neural networks: a proof-of-concept, Meas Sci Technol 28: 125301, 2017

[16] Lee Y., Yang H., Yin Z., PIV-DCNN: cascaded deep convolutional neural network for particle image velocimetry, Exp Fluids 58: 171, 2017

[17] Horn B., Schunk B., Determining optical flow, Artif Intell 17: 185-203, 1981

[18] Liu T., Merat A., Makhmalbaf H., Fajardo C., Merati P., Comparison between optical flow and cross-correlation methods for extraction of velocity fields from particle images, Exp Fluids 56: 166, 2015

[19] Dosovitskiy A., Fischer P., Ilg E., Hausser P., Hazirbas C., Golkov V., Van Der Smagt P., Cremers D., Brox T., Flownet: learning optical flow with convolutional networks, ICCV: 2758-2766, 2015

[20] Ilg E., Mayer N., Saikia T., Keuper M., Dosovitskiy A., Brox T., Flownet 2.0: evolution of optical flow estimation with deep networks, CVPR: 2462-2470, 2017

[21] Hui T., Tang X., Loi C., LiteFlowNet: a lightweight convolutional neural network for optical flow estimation, CVPR: 8981-8989, 2018

[22] Cai S., Zhou S., Xu C., Gao Q., Dense motion estimation of particle images via a convolutional neural network, Exp Fluids 60: 73, 2019

[23] Cai S., Liang J., Gao Q., Xu C., Wei R., Particle image velocimetry based on a deep learning motion estimator, IEEE T Instrum Meas: 1-1, 2019

[24] Goodfellow J., Bengio Y., Courville A., Deep learning, MIT Press, 2016

[25] Mitchell T.M., Machine learning, McGraw-Hill, 1997

[26] Ramasubramanian K., Moolayil J., Applied supervised learning with R, Pakt, 2019

[27] Vapnik V. N., The nature of statistical learning theory, Springer, 1995

[28] Fei-Fei L., Johnson J., Yeung S., CS231n*:* convolutional neural networks for visual recognition, Standford University, 2017

[29] Kingma D. P., Ba J. L., Adam: a method for stochastic optimization, 3rd International Conference for Learning Representations, ICLR, 2015.

[30] Bottou L., Stochastic gradient learning in neural networks, Proc of Neuro-Nimes, 1991

[31] Glorot X., Borders A., Bengio Y., Deep sparse rectifier neural networks, PMLR 15: 315-323, 2011

[32] He K., Zhang X., Ren S., Sun J., Delving deep into rectifiers: surpassing human-level performance on ImageNet classification, ICCV, 2015

[33] Hornik K., Stinchcombe M., White H., Multilayer feedforward networks are universal approximators, Neural Networks 2: 359-366, 1989

[34] Goodfellow I., On distinguishability criteria for estimating generative models, ICRL, 2014

[35] Karpathy A., A recipe for training neural networks, Andrej Karpathy blog, http://karpathy.github.io, 2019

[36] X. Glorot, Y. Bengio, Understanding the difficulty of training deep feedforward neural networks, PMLR 9: 249-256, 2010

[37] Ioffe S., Szegedy C., Batch normalization: accelerating deep network training by reducing internal covariate shift, ICML 37: 448-456 2015

[38] Santurkan S., Tsipras D., Ilyas A., Madry A., How does batch normalization help optimization?, NIPS: 2488-2498, 2018

[39] Yaseen A. F., Saud L. J., A survey on the layers of convolutional neural networks, IJCSMC 7: 191-196, 2018

[40] Le Cun Y., Bottou L., Bengio J., Haffner P., Gradient-based learning applied to document recognition, Proc of the IEEE 86: 2278-2324, 1998

[41] Kiger K., Introduction of PIV, Burgers Program for Fluid Dynamics Turbulence School, University of Maryland, 2010

[42] Andor Technology, Using PIV mode for iStar sCMOS camera, http://andor.oxinst.com, downloaded 2019

[43] Utami T., Blackwelder R. F., Ueno T., A cross-correlation technique for velocity field extraction from particulate visualization, Exp Fluids 10: 213-223, 1991

[44] Willert C. E., Gharib M., Digital particle image velocimetry, Exp Fluids 10: 181-193, 1991

[45] Keane R. D., Adrian R. J., Theory and simulation of particle image velocimetry, Proc. SPIE 2052, 1993

[46] Pust O., PIV: Direct cross-correlation compared with FFT-based cross-correlation, Proc of Intl Symp Appl Laser Tech Fluid Mech, 2000

[47] Computational resources were provided by HPC@POLITO (http://hpc.polito.it)

[48] The full database of particle images is available at http://www.pivchallenge.org

[49] Thielicke W., Stamhuis E. J., PIVlab – Towards user-friendly, affordable and accurate digital particle image velocimetry in Matlab, J Open Res Soft 2: 30, 2014