A supervised neural network for drag prediction of arbitrary 2D shapes in low Reynolds number flows

Jonathan Viquerat\textsuperscript{a,}\textsuperscript{*}, Elie Hachem\textsuperscript{a}

\textsuperscript{a} MINES ParisTech, PSL - Research University, CEMEF - Centre for material forming, CNRS UMR 7635, CS 10207 rue ClaudeDaunesse, 06904 Sophia-Antipolis Cedex, France.

Abstract

Despite the significant breakthrough of neural networks in the last few years, their spreading in the field of computational fluid dynamics is very recent, and many applications remain to explore. In this paper, we present a tailored VGG-like neural network architecture for low-Reynolds number flows past arbitrary 2D shapes. A set of random shapes exhibiting a rich variety of geometrical features is built using Bézier curves. The efficient labelling of the shapes is provided using an immersed method to solve a unified fluid-solid Navier-Stokes equation. The network is then trained on the obtained dataset, and its predictive efficiency assessed on several unseen shapes, including NACA airfoils.

Keywords: machine learning, neural networks, convolutional networks, computational fluid dynamics, immersed method

1. Introduction

The recent successes of machine learning (ML), and more specifically neural networks (NN), have drawn increasing attention from the scientific community on the capabilities of such methods, and their possible applications to diverse research fields. In the computational fluid dynamics (CFD) field, the topic triggered a real enthusiasm from the year 2015, with a highly-increasing amount of related papers since (see figure\textsuperscript{1}). Despite this recent hype, much remains to be done before the possibilities and limits of such methods are well contoured.

In the recent years, neural networks have been used in very different ways in order to assist or improve CFD computations. Very often, a NN is used to replace one step of the resolution process, either to attain better performance or to extricate from a limited model and gain in generality. Examples for these applications are the replacement of the pressure projection step in Eulerian methods\textsuperscript{1}, or the prediction of closure terms in RANS\textsuperscript{2} \textsuperscript{3} or LES\textsuperscript{4} computations. Direct solving of Navier-Stokes equations can also be performed with NN using physics informed deep learning, where two networks are used concurrently\textsuperscript{5} \textsuperscript{6}. The first one is trained to predict the partial differential equation (PDE) solution, while the second one is used to incorporate constraints from the original PDE.

In other cases, a flow profile or a figure of merit (such as drag or lift) can be directly seeked from a supervised network. In\textsuperscript{7}, the authors focus on the prediction of lift for 2D airfoil profiles in different flow conditions and at different incidence angles. A key point of the latter contribution is the exploration of an original method of inputting flow conditions along with airfoil profile using an "artificial image", where free-space pixels around the airfoil are colored depending on the value of the Mach number. In\textsuperscript{8}, convolutional neural networks (CNN) are trained to make a visual prediction of the steady state flow around elementary and real-life shapes, such as cars. In\textsuperscript{9}, the authors explore the capabilities of a NN to

\textsuperscript{*} Corresponding author

Email address: jonathan.viquerat@mines-paristech.fr (Jonathan Viquerat)
map design parameters of bluff bodies to flow parameters, using a stochastic gradient descent method with momentum.

In this paper, we explore the predictive capabilities of a specific neural networks architecture at low Reynolds regime around 2D randomly generated shapes. In the first section, a brief overview of the general functioning of supervised NN is presented. Then, the dataset generation is addressed, along with the efficient resolution of the Navier-Stokes equations using an embedded mesh method. In the fourth section, a baseline convolutional network is introduced and optimized. Finally, the predictive capabilities of the network are explored on unseen configurations, such as geometrical shapes or airfoils.

![Figure 1: Number of publications matching keywords "machine learning", "neural networks" and "computational fluid dynamics" in Google Scholar, between 2000 and 2018.](image)

2. Neural networks

Fundamentally, a neural network aims at approximating a function \( f: V \rightarrow W \) that represents a complex and possibly implicit relation between two spaces of finite dimensions. In supervised learning, the network is exposed to a large set of couples \((x \in V, y \in W)\) which are known to verify the relation \( y = f(x) \). For each couple, the network takes \( x \) as an input, and outputs a prediction \( y^* \). The error between \( y^* \) and \( y \) is computed, and fed back to the network, which internal parameters are adjusted accordingly via an optimization algorithm.

Classically, the tasks performed by neural networks are of two main kinds: (i) classification (e.g., analyzing handwritten text) or (ii) regression (e.g., predicting the lift of an airfoil from its shape). The goals of this paper fall under the second category. In the remaining of this section, we provide a brief description of the functioning of neural networks under supervised learning. Along the way, references to more thorough developments are also given.

2.1. Artificial neurons

The basic unit of NN is the neuron, which representation is given in figure 2. An input vector \( x \), associated to a set of weights \( w \), is provided to the neuron. The neuron then computes the weighted sum \( w \cdot x + b \), where \( b \) is called the bias, and applies the activation function \( \sigma \) to this sum. This is the output of the neuron, hereafter noted \( z \). In the neuron, the weights and the bias represent the degrees of freedom (i.e. the parameters that can be adjusted to approximate the function \( f \)), while the activation function is a hyperparameter, i.e. it is part of the choices made during the network design.
2.2. Fully connected and convolutional networks

In their simplest form, neural networks consist in several layers of neurons connected together, as shown in the basic example of figure 3. This network is said to be fully connected (FC), in the sense that each neuron of a layer is connected to all the neurons of the following layer. As it was said in last section, each connection holds a weight, and each neuron a bias, except for the input layer. Indeed, as a convention, the input layer is usually drawn as a regular layer, but it does not hold biases, nor is an activation function applied at its output. Hence, this network contains $3 \times 4$ weights and 4 biases for the hidden layer, and 4 weights and 1 bias for the output layer, for a total of 21 degrees of freedom. Along with the choice of the activation functions, the number and size of layers are also part of the hyperparameters.

When working with images as input (which will be the case in the following study), it is customary to exploit convolutional layers instead of FC ones (see figure 8). Rather than looking for patterns in their entire input space as FC layers, convolutional ones are able to extract local features. Additionally, they have the ability to build a hierarchy of increasingly complex features. In such layers, a convolution kernel (i.e. a tensor product with a weight matrix) is applied on a small patch of the input image, and is used as input for a neuron of the next layer. The patch is then slid and the operation repeated, until the input image has been entirely covered. The size of the patch is usually known as the kernel size. The entire coverage of the image with this process generates a kernel, also called filter or feature map. In most cases, multiple kernels are generated at each layer, each encoding a specific feature of the input image. Convolutional layers are often followed by pooling layers, which bring two benefits: (i) they singularly decrease the number of degrees of freedom in the network, and (ii) they help spreading the initial data throughout the successive

Figure 2: Representation of a single artificial neuron. The neuron receives an input vector $\mathbf{x}$, assorted with a weight vector $\mathbf{w}$. The output is computed as $\mathbf{w} \cdot \mathbf{x}$ corrected with the bias $b$, to which is applied the activation function $\sigma$.

Figure 3: Simple example of neural network with an input vector $\mathbf{x} \in \mathbb{R}^3$, a hidden layer composed of 4 neurons, and an output layer composed of a single neuron. As a convention, input variables are drawn using a neuron representation. However, it must be kept in mind that the input layer is not composed of neurons.
convolutional layers. When used for regression applications, convolutional networks (or convnets) most often end with a fully connected layer, followed by the output layer, which size is determined by that of the sought quantity of interest. These considerations are discussed in details in a large variety of books and articles. For the sake of brevity, we refer the reader to [10] and the references therein for complementary informations.

2.3. Technicalities

This section briefly addresses several keypoints of neural networks that will be used in the remaining of the paper. Again, this barely represents an overview of these questions, and the reader is once again referred to [10] for a thorough discussion of each of them.

2.3.1. Data pre-processing

The pre-processing of data fed to neural networks is crucial, in the sense that it may significantly influence its ability to learn. In the following, inputs are composed of images of $p \times p$ pixels with one channel (black and white image), the pixel values ranging from 0 to 255. During the pre-processing step, these images are downscaled to $n \times n$ pixels (with $n \leq p$), and the pixel values are rescaled between 0 and 1. The reason behind this normalization is that feeding large (and inhomogeneous) values to a network can prevent the gradient descent of the back-propagation method to converge [11].

Most often, the input dataset is split in three subsets: (i) a training set, on which the learning will be performed, (ii) a validation set, which is used to monitor the network accuracy periodically during training, and (iii) a test set, on which the final performance of the network is assessed. The validation and test sets must not overlap with the training set, nor between them.

2.3.2. Activation functions

Activation functions are used to obtain a non-linear mapping between the input and the output spaces. They are commonly chosen layer-wise. For classification cases, it is common to use sigmoids or hyperbolic tangents in the hidden layers, as they will stretch the input space around a central point, thus helping to separate elements from different classes. For regression cases, the rectified linear units activation function (also called relu) has proven to be a robust choice.

2.3.3. Loss function and backpropagation

The learning process in neural networks consists in adjusting all the biases and weights of the network in order to reduce the value of a well-chosen loss function. For regression cases, it is common to choose the mean squared error. With the loss function at hand, the optimization of the weights and biases is performed with a stochastic gradient descent (SGD):

1. A small subset $X$ of input elements, called batch, is randomly drawn from the input space $V$;
2. The network is run to make predictions for all elements in $X$;
3. The loss function is computed over the batch elements from the error between the prediction and the expected output;
4. The gradient of the loss with respect to the weights and biases is computed using a method called back-propagation;
5. The parameters of the network are updated so as to minimize the loss. The step size corresponding to the update is controlled by a hyperparameter called learning rate.

The back-propagation algorithm is based on the chain rule, and is at the core of the learning process, since it allows to compute the contribution of each degree of freedom of the network to the loss value.
2.3.4. Overfitting and regularization

During the training step, the network is exposed multiple times to the same input data. A full iteration over all the input samples is called an epoch, and it is not rare for advanced networks to be trained for hundreds or thousands of epochs. Inside each epoch, the network is exposed to random batches of input data, a step of SGD being performed after each batch.

If the network is trained for too long, it may start to learn features that are not part of a general trend, but that are specific to the training data. As a consequence, the network will perform better and better on training data after each epoch, but its performance on the validation data will stabilize (at best) or decrease. This phenomenon, called overfitting, is highly dependent on the complexity of the network (i.e. the number and size of layers), the amount of available input data, and the complexity of the features one is trying to make the network learn.

Several methods are available to limit overfitting. The first one consists in gathering more data, although this is often either impossible or expensive. A second option is to reduce the size of the network, in order to better balance the number of free parameters with the size of available input data. However, this may also lead to a loss in the generalization capabilities. The last option consists in using regularization, which can be done in two ways:

1. A penalization term proportional to the squared weights ($l^2$ regularization) or their absolute value ($l^1$ regularization) can be applied to the loss function. This will globally constrain the weights to be smaller, which will favor the emergence of simpler features over complex (and specific) ones;
2. A dropout layer can also be applied between two hidden layers: this consists in randomly setting to zero a fraction of the information passing from one layer to the next. The goal is to introduce some random noise in the information travelling through the network in order to prevent fortuitous patterns to be learned.

2.3.5. Neural network libraries

The amount of ready-to-use neural networks libraries has exploded in the recent years, most of them exploiting C++ or Python. For supervised learning, they usually include a wide range of choices regarding layer types, activation functions, losses, optimizers and so on. In this paper, we chose to use Keras [11] for its high level of abstraction and the ease of use provided by the Python language.

3. Dataset generation

This section describes the generation of the dataset used in the remaining of the paper. First, we describe the steps to generate arbitrary shapes by means of connected Bezier curves. Then, the solving of the Navier-Stokes equations with an immersed method is presented. Finally, details about the dataset are given.

3.1. Random shape generation

The first step of the random shape generation consists in drawing $n_s$ random points in $[0, 1]^2$, that are then translated so their center of mass is in $(0, 0)$. The points are then sorted by ascending trigonometric angle (see figure 4a). The angles between consecutive random points are then computed, and an average is computed around each point (see figure 4b):

$$\theta_i^* = \alpha \theta_{i-1} + (1 - \alpha) \theta_{i+1},$$

with $\alpha \in [0, 1]$. Averaging angles in such way will help smooth the final obtained shape. In the next step, a third order Bézier curve is drawn between each point, using the averaged angles $\theta_i^*$. Cubic Bézier curves are defined by four points: the first and last points, $p_i$ and $p_{i+1}$, are part of the curve, while the second and third ones, $p_i^*$ and $p_{i+1}^*$, are control points that define the tangent of the curve at $p_i$ and $p_{i+1}$. In our case, the tangents at $p_i$ and $p_{i+1}$ are determined respectively by $\theta_i^*$ and $\theta_{i+1}^*$ (see figure 4c). In a final step, all the
Beziers curves are sampled, and a closed loop is exported to be used as an immersed mesh in a Navier-Stokes numerical simulation (figure 4d).

The sharpness of the curve features is handled with a positive parameter $r$ that controls the distances $[p_ip_i^*]$ and $[p_{i+1}p_{i+1}^*]$. For $r = 0$, $p_i^*$ and $p_{i+1}^*$ respectively coincide with $p_i$ and $p_{i+1}$, and the curve presents sharp angles at the control points. Intermediate values of $r$ produce smooth curves, with maximal smoothness for $r = 0.5$. When increasing further toward $r = 1$, sharp features start to appear near the crossing of the initial and final curve tangents. Finally, for $r > 1$, tangled cases start to appear. A variety of shapes obtained with different values of $r$ can be found in figure 5. In the following, we restrict $r$ to the interval $[0, 1]$ to avoid tangled shapes.

### 3.2. Navier-Stokes equations

The flow motion of incompressible Newtonian fluids is described by the Navier-Stokes (NS) equations:

**Figure 4:** Random shape generation with cubic Bézier curves.
Figure 5: Random shape examples depending on their $r$ value, ranging from 0 to 1. The random points are shown in blue, and their number $n_s$ ranges from 3 to 5, although it is possible to use more. For $r = 0$, one sees the sharp features of the curve on the Bézier points. For intermediate values, smooth curves are obtained. Finally, for values close to 1, sharp features start to appear around the control points (not shown here).

\[
\begin{align*}
\rho \left( \partial_t \mathbf{v} + \mathbf{v} \cdot \nabla \mathbf{v} \right) - \nabla \cdot \left( 2\eta \varepsilon(\mathbf{v}) - p \mathbf{I} \right) &= \mathbf{f}, \\
\nabla \cdot \mathbf{v} &= 0,
\end{align*}
\]

where $t \in [0,T]$ is the time, $\mathbf{v}(x,t)$ the velocity, $p(x,t)$ the pressure, $\rho$ the fluid density, $\eta$ the dynamic viscosity and $\mathbf{I}$ the identity tensor. Classically, the solving of NS equations around solid obstacles relies on body-fitted methods, where the mesh boundary follows the geometry of the obstacle. These methods require the generation of a full mesh (domain and obstacle) for each computation, which can be both time- and memory-consuming (see figure 6a). To overcome this issue, immersed methods based on a unified fluid-solid eulerian formulation were introduced that propose to immerse a boundary mesh of the obstacle in a background mesh (see figure 6b). The outline of this method is sketched in the next section.

3.3. Interface description

The formulation presented in this section is based on the introduction of an extra stress in the momentum equation of 1. This extra stress is related to the appropriate deformation tensor in the solid domain and acts as a Lagrange multiplier to enforce that the deformation be zero in the solid.

In the fluid-structure interaction field, monolithic approaches impose the use of an appropriate constitutive equation describing both the fluid and the solid domain. This offers a great flexibility to deal with different shapes in similar configurations without having to systematically re-mesh the whole domain. To do so, one starts by computing the signed distance function (level set) of the given geometry to each node of the background mesh:

\[
\alpha(x) = \pm d(x, \Gamma_{im}), \forall x \in \Omega.
\]
Using the zero iso-value of this function, the fluid-solid interface $\Gamma_{\text{im}}$ is easily identified as the zero iso-value of function $\alpha$:

$$\Gamma_{\text{im}} = \{ \mathbf{x} \in \Omega, \alpha(\mathbf{x}) = 0 \}.$$  \hfill (3)

In this paper, the following sign convention is used: $\alpha \geq 0$ inside the solid domain defined by the interface $\Gamma_{\text{im}}$, and $\alpha \leq 0$ outside this domain. Further details about the algorithm used to compute the distance are available in [12]. It is also possible to use functions smoother than $d(\mathbf{x}, \Gamma_{\text{im}})$ away from $\Gamma_{\text{im}}$ (see for example [13]).

As explained above, the signed distance function is used to localize the interface of the immersed structure, but it is also used to initialize the desirable properties on both sides of the latter. Indeed, for the elements crossed by the level-set functions, fluid-solid mixtures are used to determine the element effective properties. To do so, a Heaviside function $H(\alpha)$ is defined as follows:

$$H(\alpha) = \begin{cases} 
1 & \text{if } \alpha > 0, \\
0 & \text{if } \alpha < 0.
\end{cases}$$  \hfill (4)

The Heaviside function can be smoothed to obtain a better continuity at the interface [14] using the following expression:

$$H_{\varepsilon}(\alpha) = \begin{cases} 
1 & \text{if } \alpha > \varepsilon, \\
\frac{1}{2} \left(1 + \frac{\alpha}{\varepsilon} + \frac{1}{2} \sin \left(\frac{\pi \alpha}{\varepsilon}\right)\right) & \text{if } |\alpha| \leq \varepsilon, \\
0 & \text{if } \alpha < -\varepsilon,
\end{cases}$$  \hfill (5)

where $\varepsilon$ is a small parameter such that $\varepsilon = O(h_{\text{im}})$, known as the interface thickness, and $h_{\text{im}}$ is the mesh size in the normal direction to the interface.

3.4. Modified governing equations

Now that each system is expressed in an eulerian framework, we solve one global NS set of equations using the geometrical representation given by $H(\alpha)$ as follows:

$$\begin{cases} 
\rho^*(\partial_t \mathbf{v} + \mathbf{v} \cdot \nabla \mathbf{v}) - \nabla \cdot (2\eta \epsilon(\mathbf{v}) + \mathbf{\tau} - p \mathbf{I}) = \mathbf{f}, \\
\nabla \cdot \mathbf{v} = 0,
\end{cases}$$  \hfill (6)

where we have introduced the following mixed quantities:

$$\mathbf{\tau} = H(\alpha) \mathbf{\tau}_s,$$
$$\rho^* = H(\alpha) \rho_s + (1 - H(\alpha)) \rho_f,$$

the subscripts $f$ and $s$ referring respectively to the fluid and to the solid. In the latter equalities, $\mathbf{\tau}_s$ acts as a Lagrange multiplier that yields $\epsilon(\mathbf{v}) = 0$ in the solid [15].

Eventually, the modified equations (6) are cast into a stabilized finite element formulation, and solved using a variational multi-scale (VMS) solver (the reader is invited to refer to [15] for more details).

3.5. Dataset

The dataset (DS) is composed of 12,000 shapes, along with their steady-state drag value at $Re = 10$ (see figure 7). To ensure a large diversity of shapes, $n_s$ is evenly distributed in $[3, 5]$, and $r$ in $[0, 1]$. All the latter labels were computed using cimLib [15], following the methods exposed in sections 3.2, 3.3 and 3.4. In the following, the DS is systematically divided into three sets: 9600 shapes for the training set, 1200 shapes for the validation set, and 1200 shapes for the test set.
4. Results

4.1. Baseline network

Here, we introduce a baseline network with reasonable performance on the drag prediction task, and which optimization will be the topic of the next sections. The chosen architecture is similar to that of the Visual Geometry Group (VGG) [10], which was initially designed for image classification. In this network, a convolution/pooling pattern is repeated several times, the number of convolutional filters being doubled after each pooling layer (see figure 8). In our case, the number of convolution layers in the base pattern is set to 2, with 8 filters in the initial layer. The network is terminated with 2 dense layers of size 256. The last layer is used to output the predicted drag and is therefore of size 1, and uses a linear activation function, while all the other layers of the network use ReLU activations. For the baseline case, the input images were downscaled with a factor \(s = 4\) (see section 4.2). In the remaining of the paper, the learning rate is set to \(1 \times 10^{-4}\), and the total number of epochs to 100 to avoid overfitting. For a better reproducibility, the random seeds are set to the same value throughout all the following numerical experiments.

Figure 8: **Example of drag prediction convolutional network**, which architecture is similar to VGG. This network is based on a pattern made of two convolutional layers (in light blue) followed by a max-pooling layer (in orange), with an original input of size \(n \times n\) (possibly downscaled by a factor \(s\)). The pattern is repeated four times, the image size being divided by two each time, as the number of filters doubles. The last max-pooling layer is followed by two fully-connected layers (in gray). It is terminated with a fully-connected layer of size 1 that outputs the predicted drag through a linear activation function.
The performance of the network is computed by measuring the relative drag prediction error on the test subset. To do so, a forward network pass is made for each shape of the subset to obtain the predicted drag, which is compared to the exact drag. The relative prediction error is then computed. A plot of the error levels on the test set is shown in figure 9. The low average relative error indicates a good overall accuracy, except for some shapes presenting a low drag (roughly, $C_x \leq 1$), for which levels as high as 5% can be reached. As could be expected, the worst performing shapes are those with the smallest areas (see figure 9).

4.2. Input downscaling

At maximum resolution, the size of the input images provided to the network is 256 pixels. However, it is common to downscale input images, which naturally reduces the complexity of the network as well as the cost of the training. Examples of downscaled inputs are shown in figure 10. Too severe downscaling may affect the ability of the network to properly identify the shape contours, and thus to correctly infer the resulting drag. In figure 11 we evaluate the impact of the downscaling factor $s$ on the final network performance. $s = 4$ represents an adequate choice to balance the accuracy of the trained network with the learning cost, and will be used in the remaining of the paper.

![Figure 9: Relative error obtained for drag prediction on the test set with baseline network architecture. The horizontal bar indicates the average relative error level.](image)

![Figure 10: Original and downscaled images used as input of the convolutional network. Downscaling input images allows to reduce the number of degrees of freedom used in the network, hence speeding up the learning. However, it may also make it more difficult for the network to catch the features of the shape, thus making learning less accurate. Here $s$ corresponds to the downscaling factor.](image)
4.3. Batch size

The choice of the batch size in supervised learning is known to have a major impact on the performances of the resulting network [10]. It has multiple outcomes, such as (i) the accuracy of the gradient estimate, (ii) the time required for training or (iii) the necessity of an adequate learning rate. In figure 12, we plot the average and maximal relative prediction errors as a function of the batch size using the baseline network of section 4.1. As can be seen, using a batch size of size 16 provides a neat improvement over the other choices, both for average and maximal error. The relative error over the test subset is also shown in figure 12 as can be seen, the high prediction errors obtained for the smaller-sized shapes in figure 9 have now dropped down to a level similar to that of other shapes of larger sizes, with a maximal error level as low as 2.6%.

Figure 11: Analysis of the influence of input downscaling on the average and maximal relative error. For $s \in [1, 4]$, the average and maximal error levels rise slowly with the downscaling factor. Above this value, the maximal error level rises suddenly, indicating that a resolution threshold was passed, below which some shape features become indiscernible by the convolution kernels. The baseline case is indicated with dark dots.

Figure 12: Analysis of the influence of batch size on the average and maximal relative error. A neat improvement is obtained using a batch size of 16.
4.4. Fully-connected layers

In this section, the possibility of improving the network architecture further by modifying the width and depth of the fully connected layers is explored. In convolutional networks, the task attributed to these layers is to learn non-linear combinations of the high-level features extracted by the convolutional blocks. Modern classification networks such as VGG [16] or ResNet [17] usually include zero to a few dense layers between the convolutional blocks and the output layer. In figure 13, the average and maximal relative errors of the network when using different structures for the dense layers are shown. Although moving to deeper fully-connected stacked seems to slightly improve the accuracy of the network, it also significantly increases its complexity and its learning cost. For that reason, we consider the use of two 256-wide dense layers as a good local optimum for the remaining of the paper.

![Figure 13: Analysis of fully-connected layers architecture on the average and maximal relative error. Using two layers of 256 neurons represents a good balance between the network complexity and the accuracy, although a deeper stack of dense layers seem to improve further both the average and the maximal relative errors.](image)

4.5. Drag prediction on real-life shapes

We now evaluate the predictive capabilities of the best network on a selected set of shapes, including geometrical shapes (cylinder, square) and NACA airfoils. The results are summed up in table 1. The shapes dimensions are adapted to fit the mean dimensions of the dataset shapes, i.e. they fit in the $[-1,1]^2$ square, with their center of mass centered in $(0,0)$. Relative error levels remain low on such shapes, with a maximal value of 2.53% for the horizontal bar. As could be expected, the prediction on a shape of the dataset yields a very accurate result, with a relative error below 0.1%. Finally, the error levels for NACA airfoils remain low, with a maximum level of 2%. This experiment also underlines the interest of the random shape dataset for the drag prediction on non-random, real-life shapes.

4.6. Conclusion

In this paper, a tailored VGG-like neural network was introduced for the drag prediction of arbitrary 2D shapes in laminar flow at $Re = 10$. This network was trained on a custom dataset composed of 12,000 random shapes built with Bézier curves, and which drag was computed numerically solving the Navier-Stokes equations using an immersed mesh method. The large variety of geometrical shapes in the dataset allowed the network to make accurate drag predictions on unseen shapes such as NACA airfoils, with a maximal relative error in the 1-2% range.

These results underline the potential of this approach, and shall be pursued at higher Reynolds. Also, exploiting advanced network architectures, such as ResNets or densely connected convolutional networks, may lead to even better results.
Table 1: **Exact and predicted drags for several handpicked shapes.** The shapes largest dimensions were adapted to fit the mean dimensions of the dataset shapes. The different geometrical parameters given in the array are the following: \( w \) stands for width, \( h \) stands for height, \( r \) stands for radius and \( c \) stands for chord. It is important to notice that in the following table, the scale of the NACA airfoils is not that of the other shapes.

| Shape | Description | Prediction (rel. error) | Exact drag |
|-------|-------------|------------------------|------------|
| ![Vertical bar](image) | \( h = 1, \) \( w = 0.2 \) | 1.611 (0.94%) | 1.596 |
| ![Horizontal bar](image) | \( h = 0.2, \) \( w = 1 \) | 0.925 (2.53%) | 0.949 |
| ![Cross](image) | \( w = 1, \) \( h = 0.2 \) | 1.579 (1.65%) | 1.553 |
| ![Cylinder](image) | \( r = 0.5 \) | 1.600 (0.69%) | 1.589 |
| ![Square](image) | \( h = w = 1 \) | 1.787 (1.31%) | 1.764 |
| ![Random shape from DS](image) | | 1.898 (0.053%) | 1.897 |
| ![NACA 0018](image) | \( c = 1 \) | 1.181 (0.42%) | 1.186 |
| ![NACA 4412](image) | \( c = 1 \) | 1.099 (2.0%) | 1.121 |
| ![NACA 4424](image) | \( c = 1 \) | 1.280 (1.34%) | 1.263 |
| ![NACA 6412](image) | \( c = 1 \) | 1.124 (0.71%) | 1.132 |

References

[1] J. Tompson, K. Schlachter, P. Sprechmann, K. Perlin, Accelerating Eulerian Fluid Simulation With Convolutional Networks, arXiv. URL http://arxiv.org/abs/1607.03597

[2] J. Ling, A. Kurzawski, J. Templeton, Reynolds averaged turbulence modelling using deep neural networks with embedded invariance, Journal of Fluid Mechanics 807 (2016) 155–166.

[3] B. D. Tracey, K. Duraisamy, J. J. Alonso, A Machine Learning Strategy to Assist Turbulence Model Development in: 53rd AIAA Aerospace Sciences Meeting, 2015, pp. 1–23. URL http://arc.aiaa.org/doi/10.2514/6.2015–1287

[4] A. D. Beck, D. G. Flad, C.-D. Munz, Deep Neural Networks for Data-Driven Turbulence Models, arXiv. URL http://arxiv.org/abs/1806.04482

[5] M. Raissi, P. Perdikaris, G. E. Karniadakis, Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations, arXiv (2017) 1–22. URL http://arxiv.org/abs/1711.10566

[6] M. Raissi, A. Yazdani, G. E. Karniadakis, Hidden Fluid Mechanics: A Navier-Stokes Informed Deep Learning Framework for Assimilating Flow Visualization Data, arXiv. URL http://arxiv.org/abs/1808.04327

[7] Y. Zhang, W.-J. Sung, D. Mavris, Application of Convolutional Neural Network to Predict Airfoil Lift Coefficient, arXiv (2017) 1–9. URL http://arxiv.org/abs/1712.10082
[8] X. Guo, W. Li, F. Iorio, Convolutional Neural Networks for Steady Flow Approximation Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD ’16 (2016) 481–490. URL http://dl.acm.org/citation.cfm?doid=2939672.2939738
[9] T. P. Miyawalwala, R. K. Jaiman, An Efficient Deep Learning Technique for the Navier-Stokes Equations: Application to Unsteady Wake Flow Dynamics arXiv (2017) 1–46. URL http://arxiv.org/abs/1710.09099
[10] A. C. Ian Goodfellow, Yoshua Bengio, The Deep Learning Book, MIT Press, 2017.
[11] F. Chollet, Deep Learning with Python, Manning, 2018.
[12] J. Bruchon, H. Digonnet, T. Coupez, Using a signed distance function for the simulation of metal forming processes: Formulation of the contact condition and mesh adaptation, International Journal for Numerical Methods in Engineering 78 (8) (2009) 980–1008.
[13] R. Codina, O. Soto, A numerical model to track two-fluid interfaces based on a stabilized finite element method and the level set technique, International Journal for Numerical Methods in Fluids 40 (2002) 293–301.
[14] S. van der Pijl, A. Segal, C. Vuik, P. Wesseling, A mass-conserving level-set method for modelling of multi-phase flows, International Journal for Numerical Methods in Fluids 47 (2005) 339–361.
[15] E. Hachem, S. Pehlali, R. Codina, T. Coupez, Immerged stress method for fluid structure interaction using anisotropic mesh adaptation, International Journal for Numerical Methods in Engineering 94 (2013) 805–825.
[16] K. Simonyan, A. Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition (2014) 1–14. URL http://arxiv.org/abs/1409.1556
[17] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition 2016-December (2016) 770–778.