Computers and Fluids 225 (2021) 104973

Contents lists available at ScienceDirect

Computers and Fluids

journal homepage: www.elsevier.com/locate/compfluid

A review on deep reinforcement learning for fluid mechanics

Paul Garnier[a], Jonathan Viquerat[a], Jean Rabault[b], Aurélien Larcher[a], Alexander Kuhnle[c], Elie Hachem[a,*]

[a] MINES ParisTech, PSL Research University, Centre de mise en forme des matériaux (CEMEF), CNRS UMR 7635, 06904 Sophia Antipolis Cedex, France
[b] Department of Mathematics, University of Oslo, Oslo, 0851, Norway
[c] Department of Computer Science and Technology, University of Cambridge, United Kingdom

ARTICLE INFO

Article history:
Received 12 August 2019
Revised 26 February 2021
Accepted 12 April 2021
Available online 1 May 2021

Keywords:
Deep reinforcement learning
Fluid mechanics

ABSTRACT

Deep reinforcement learning (DRL) has recently been adopted in a wide range of physics and engineering domains for its ability to solve decision-making problems that were previously out of reach due to a combination of non-linearity and high dimensionality. In the last few years, it has spread in the field of computational mechanics, and particularly in fluid dynamics, with recent applications in flow control and shape optimization. In this work, we conduct a detailed review of existing DRL applications to fluid mechanics problems. In addition, we present recent results that further illustrate the potential of DRL in Fluid Mechanics. The coupling methods used in each case are covered, detailing their advantages and limitations. Our review also focuses on the comparison with classical methods for optimal control and optimization. Finally, several test cases are described that illustrate recent progress made in this field. The goal of this publication is to provide an understanding of DRL capabilities along with state-of-the-art applications in fluid dynamics to researchers wishing to address new problems with these methods.

1. Introduction

The process of learning and how the brain understands and classifies information has always been a source of fascination for humankind. Since the beginning of research in the field of artificial intelligence (AI), an algorithm able to learn to make decisions on its own has been one of the ultimate goals. Going back to the end of the 1980s [52], reinforcement learning (RL) proposed a formal framework in which an agent learns by interacting with an environment through the gathering of experience [14]. Significant results were obtained during the following decade [55], although they were limited to low-dimensional problems.

In recent years, the rise of deep neural networks (DNN) has provided reinforcement learning with new powerful tools [9,17,51]. The combination of deep learning with RL, called deep reinforcement learning (DRL), lifted several major obstacles that hindered classical RL by allowing the use of high-dimensional state spaces and exploiting the feature extraction capabilities of DNNs. Recently, DRL managed to perform tasks with unprecedented efficiency in many domains such as robotics [41] or language processing [4], and even achieved superhuman levels in multiple games: Atari games [36], Go [50], Dota II [40], Starcraft II [59] and even Poker [11]. Also in industrial application, DRL has proven itself useful. For example, *Wayve* trains autonomous cars both onboard [25] and through simulation [8], and Google uses DRL in order to control the cooling of its data centers [26].

In the field of fluid mechanics and mechanical engineering, one is also confronted with nonlinear problems of high dimensionality. For example, using computational simulations to test several different designs or configurations has proven a useful technique. However, the number of possibilities to explore can make such search difficult since it is often unfeasible to evaluate all configurations exhaustively. Therefore, the assistance of automatic optimization procedures is needed to help find optimal designs.

Possible applications range from multiphase flows in microfluidics to shape optimization in aerodynamics or conjugate heat transfer in heat exchangers, and many other fields can gain from such techniques such as biomechanics, energy, or marine technologies. This is the primary motivation for the combination of computational fluid dynamics (CFD) with numerical optimization methods. Most of the time, the main obstacle is the high computational cost of determining the sensitivity of the objective function to variations of the design parameters by repeated calculations of the flow. Besides, classical optimization techniques, like direct gradient-based methods, are known for their lack of robust-

* Corresponding author.
E-mail addresses: paul.garnier@mines-paristech.fr (P. Garnier), jonathan.viquerat@mines-paristech.fr (J. Viquerat), jean.rblt@gmail.com (J. Rabault), aurelien.larcherm@mines-paristech.fr (A. Larcher), alexkuhnle@t-online.de (A. Kuhnle), elie.hachem@mines-paristech.fr (E. Hachem).

https://doi.org/10.1016/j.compfluid.2021.104973
0045-7930/

ness and for their tendency to fall into local optima. Generic and robust search methods, such as evolutionary or genetic algorithms, offer several attractive features and have been used widely for design shape optimization [1,27,33]. In particular, they can be used for multi-objective multi-parameter problems. They have been successfully tested for many practical cases, for example for design shape optimization in the aerospace [34] and automotive industries [38]. However, the use of a fully automatic evolutionary algorithms coupled with CFD for the optimization of multi-objective problems remains limited by the computing burden at stake and is still far from providing a practical tool for many engineering applications.

Despite great potential, the literature applying DRL to computational or experimental fluid dynamics remains limited. In recent years, only a handful of applications were proposed that take advantage of such learning methods. These applications include flow control problems [18,43], process optimization [28,39], shape optimization [3], and our own results about laminar flows past a square cylinder presented later in this work, among others.[1]

This paper presents an introduction to modern RL for the neophyte reader to get a grasp of the main concepts and methods existing in the domain, including the necessary basics of deep learning. Subsequently, the relevant fluid mechanics concepts are reiterated. We then go through a thorough review of the literature and describe the content of each contribution. In particular, we cover the choice of DRL algorithm, the problem complexity, and the concept of reward shaping. Finally, an outline is given on the future possibilities of DRL coupled with fluid dynamics.

2. Deep reinforcement learning

In this section, the basic concepts of DRL are introduced. For the sake of brevity, some topics are only minimally covered, and the reader is referred to more detailed introductions to (D)RL [14,54]. First, RL is presented in its mathematical description as a Markov Decision Process. Then, its combination with deep learning is detailed, and an overview of several DRL algorithms is presented.

2.1. Reinforcement learning

At its core, reinforcement learning models an agent interacting with an environment and receiving observations and rewards from it, as shown in Fig. 1. The goal of the agent is to determine the best action in any given state in order to maximize its cumulative reward over an episode, *i.e.* over one instance of the scenario in which the agent takes actions. Taking the illustrative example of a board game:

1. An episode is equivalent to one game;
2. A state consists of the summary of all pieces and their positions;
3. Possible actions correspond to moving a piece of the game;
4. The reward may simply be $+1$ if the agent wins, -1 if it loses and 0 in case of a tie or a non-terminal game state.

Generally, the reward is a signal of how *'good'* a board situation is, which helps the agent to learn distinguish more promising from less attractive decision trajectories. A trajectory is a sequence of states and actions experienced by the agent:

$$\tau = \left(s_0, a_0, s_1, a_1, \dots\right).$$

The cumulative reward (*i.e.* the quantity to maximize) is expressed along a trajectory, and includes a discount factor $\gamma \in [0, 1]$ that smoothes the impact of temporally distant rewards (it is then called *discounted* cumulative reward):

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t.$$

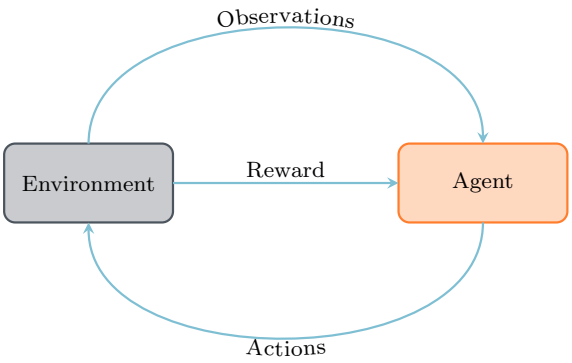


Fig. 1. DRL agent and its environment.

RL methods are classically divided in two categories, namely *model-free* and *model-based* algorithms. Model-based method incorporates a model of the environment they interact with, and will not be considered in this paper (the reader is referred to [54] and references therein for details about model-based methods). On the contrary, model-free algorithms directly interact with their environment, and are currently the most commonly used within the DRL community, mainly for their ease of application and implementation. Model-free methods are further distinguished between *value-based* methods and *policy-based* methods [53]. Although both approaches aim at maximizing their expected return, policy-based methods do so by directly optimizing the decision policy, while value-based methods learn to estimate the expected value of a state-action pair optimally, which in turn determines the best action to take in each state. Both families inherit from the formulation of Markov decision processes (MDP), which is detailed in the following section.

2.1.1. Markov decision processes

A MDP can be defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathbb{P}, R)$ [5,24], where:

1. $\mathcal{S}$, $\mathcal{A}$ and R are defined in Table 1,
2. $\mathbb{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}^+ \rightarrow [0, 1]$, where $\mathbb{P}(s'|s, a)$ is the probability of getting to state s' from state s following action a.

In the formalism of MDP, the goal is to find a policy $\pi(s)$ that maximizes the expected reward. However, in the context of RL, $\mathbb{P}$ and R are unknown to the agent, which is expected to come up with an efficient decisional process by interacting with the environment. The way the agent induces this decisional process classifies it either in value-based or policy-based methods.

2.1.2. Value-based methods

In value-based methods, the agent learns to optimally estimate a *value function*, which in turn dictates the policy of the agent by selecting the action of the highest value. One usually defines the *state value function*:

$$V^\pi(s) = \mathop{\mathbb{E}}_{\tau\sim\pi}\left[R(\tau)|s\right],$$

and the *state-action value function*:

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\tau\sim\pi}\left[R(\tau)|s, a\right],$$

[1] These references and more are addressed in details in the remaining of this article.

Table 1
Notations regarding DRL.

γ	discount factor
λ	learning rate
α, β	step-size
ϵ	probability of random action
s, s'	states
$\mathcal{S}^+$	set of all states
$\mathcal{S}$	set of non-termination states
a	action
$\mathcal{A}$	set of all actions
r	reward
$\mathcal{R}$	set of all rewards
t	time station
T	final time station
a_t	action at time t
s_t	state at time t
r_t	reward at time t
$R(\tau)$	discounted cumulative reward following trajectory τ
$R(s,a)$	reward received for taking action a in state s
π	policy
θ, θ'	parameterization vector of a policy
π_θ	policy parameterized by θ
$\pi(s)$	action probability distribution in state s following π
$\pi(a\|s)$	probability of taking action a in state s following π
$V^\pi(s)$	value of state s under policy π
$V^*(s)$	value of state s under the optimal policy
$Q^\pi(s,a)$	value of taking action a in state s under policy π
$Q^*(s,a)$	value of taking action a in state s under the optimal policy
$Q_\theta(s,a)$	estimated value of taking action a in state s with parameterization θ

which respectively denote the expected discounted cumulative reward starting in state s (resp. starting in state s and taking action a) and then follow trajectory τ according to policy π. It is fairly straightforward to see that these two concepts are linked as follows:

$$V^\pi(s) = \mathop{\mathbb{E}}_{a\sim\pi}\Big[Q^\pi(s,a)\Big],$$

meaning that in practice, $V^\pi(s)$ is the weighted average of $Q^\pi(s,a)$ over all possible actions by the probability of each action. Finally, the *state-action advantage function* can be defined as $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$. One of the main value-based methods in use is called Q-learning, as it relies on the learning of the Q-function to find an optimal policy. In classical Q-learning, the Q-function is stored in a Q-table, which is a simple array representing the estimated value of the optimal Q-function $Q^*(s,a)$ for each pair $(s,a) \in \mathcal{S}\times\mathcal{A}$. The Q-table is initialized randomly, and its values are progressively updated as the agent explores the environment, until the Bellman optimality condition [6] is reached:

$$Q^*(s,a) = R(s,a) + \gamma \max_{a'} Q^*(s',a'). \tag{1}$$

The Bellman equation indicates that the Q-table estimate of the Q-value has converged and that systematically taking the action with the highest Q-value leads to the optimal policy. In practice, the expression of the Bellman equation [6] is used to update the Q-table estimates.

2.1.3. Policy-based methods

Another approach consists of directly optimizing a parameterized policy $\pi_\theta(a|s)$, instead of resorting to a value function estimate to determine the optimal policy. In contrast with value-based methods, policy-based methods offer three main advantages:

1. They have better convergence properties, although they tend to be trapped in local minima;
2. They naturally handle high dimensional action spaces;
3. They can learn stochastic policies.

To determine how "good" a policy is, one defines an objective function based on the expected cumulative reward:

$$J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\Big[R(\tau)\Big],$$

and seeks the optimal parameterization θ^* that maximizes $J(\theta)$:

$$\theta^* = \arg\max_\theta \mathbb{E}\Big[R(\tau)\Big].$$

To that end, the gradient of the cost function is needed. This is not an obvious task at first, as one is looking for the gradient with respect to the policy parameters θ, in a context where the effects of policy changes on the state distribution are unknown. Indeed, modifying the policy will most certainly modify the set of visited states, which could affect performance in an unknown manner. This derivation is a classical exercise that relies on the log-probability trick [60], which allows expressing $\nabla_\theta J(\theta)$ as an evaluable expected value:

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log\left(\pi_\theta(a_t|s_t)\right) R(\tau)\right]. \tag{2}$$

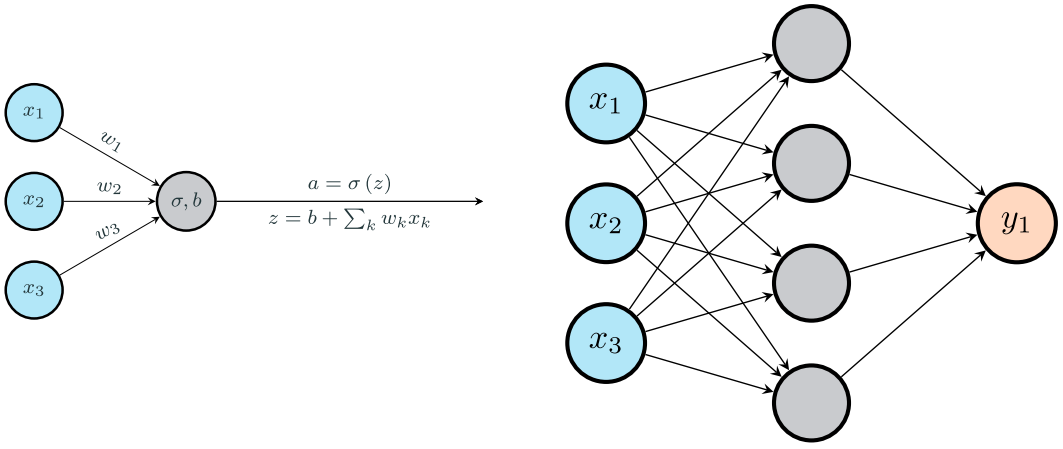
This gradient is then used to update the policy parameters:

$$\theta \leftarrow \theta + \lambda \nabla_\theta J(\theta). \tag{3}$$

As $\nabla_\theta J(\theta)$ takes the form of expected value, in practice, it is evaluated by averaging its argument over a set of trajectories. In such vanilla policy gradient methods, the actions are evaluated at the end of the episode (they belong to Monte-Carlo methods). If some low-quality actions are taken along the trajectory; their negative impact will be averaged by the high-quality actions and will remain undetected. This problem is overcome by actor-critic methods, in which a Q-function evaluation is used in conjunction with a policy optimization.

2.2. Deep reinforcement learning

In RL, the evaluation of value functions such as V^π or Q^π can be done in several ways. One possible way, as presented earlier, is



(a) Representation of a single artificial neuron. The neuron receives an input vector x, assorted with a weight vector w. The output is computed as $w \cdot x$ corrected with the bias b, to which is applied the activation function σ.

(b) Simple example of neural network with an input vector $x \in \mathbb{R}^3$, 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.

Fig. 2. Representation of a single artificial neuron and a basic 3-layer neural network.

to use tables that store the values for every state or action-state pair. However, this strategy does not scale with the sizes of state and action spaces. Another possibility is to use neural networks to estimate value functions, or, for policy-based methods, to output an action distribution given input states. Artificial neural networks can be seen as universal function approximators [49], which means that, if properly trained, they can represent arbitrarily complex mappings between spaces. This particularly suitable property of ANNs gave rise to deep reinforcement learning, *i.e.* reinforcement learning algorithms using deep neural networks as functions approximators. Regarding the notions introduced in previous sections, it is natural to think of θ as the parameterization of the neural network, *i.e.* as it set of weights and biases. In the following section, a basic introduction to neural networks is given. The reader is referred to [17] for additional details.

2.2.1. Artificial neurons and neural networks

The basic unit of a neural network (NN) is the neuron, which representation is given in Fig. 2. An input vector x, associated with 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 σ 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 hyper-parameter, *i.e.* , it is part of the choices made during the network design.

In their purest form, neural networks consist of several layers of neurons connected to each other, as shown in the primary example of 2. Such a 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 the last section, each connection is characterized by a weight, and in addition each neuron has 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. 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 that represents the quality of the network prediction. This update is usually performed by a stochastic gradient method, in which the gradients of the loss function with respect to the weights and biases are estimated using a back-propagation algorithm.

When working for instance with images as input, it is customary to exploit convolutional layers instead of FC ones. Rather than looking for patterns in their entire input space as FC layers, convolutional ones can extract local features. Additionally, they can build a hierarchy of increasingly sophisticated 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 moved, 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 complete 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. 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 such as [9,17,51].

2.3. DRL algorithms

In this section, the main lines of the major DRL algorithms in use today are presented. For the sake of brevity, many details and discussions are absent from the following discussions. The reader is referred to the profuse literature on the topic.

2.3.1. Deep Q-networks

Instead of updating a table holding Q-values for each possible (s, a) pair, ANNs can be used to generate a map $\mathcal{S}^+ \times \mathcal{A} \longrightarrow \mathbb{R}$ (called deep Q-networks (DQN)) that provides an estimate of the Q-value for each possible action given an input state. As it is usual for neural networks, the update of the Q-network requires a loss function for the gradient descent algorithm, that will be used to optimize the network parameters θ. To do so, it is usual to exploit

the Bellman optimality Eq. (1):

$$L(\theta) = \mathbb{E}\left[\frac{1}{2}\Big(\underbrace{\big[R(s,a) + \gamma \max_{a'} Q_\theta(s',a')\big]}_{\text{Target}} - Q_\theta(s,a)\Big)^2\right]. \tag{4}$$

In the latter expression, $Q_\theta(s,a)$ represents the Q-value *estimate* provided by the DQN for action s and state a under network parameterization θ. It must be noted that the quantity denoted *target* is the same that appears in the Bellman optimality Eq. (1): when the optimal set of parameters θ^* is reached, $Q_\theta(s,a)$ is equal to the target, and $L(\theta)$ is equal to zero.

In vanilla deep Q-learning (and more generally in Q-learning), an exploration/exploitation trade-off must be implemented, to ensure a sufficient exploration of the environment. To do so, a parameter $\epsilon \in [0,1]$ is defined, and a random value is drawn in the same range before each action. If this value is below ϵ, a random action is taken. Otherwise, the algorithm follows the action prescribed by $\max_a Q_\theta(s,a)$. Most often, ϵ follows a schedule, starting with high values at the beginning of learning, and progressively decreasing. The vanilla algorithm using DQN is shown below:

Algorithm 1 Vanilla deep Q-learning.

Initialize action-value function parameters θ_0
for $k = 0, M$ **do** ▷ Loop over episodes
  for $t = 0, T$ **do** ▷ Loop over time stations
    Draw random value $\omega \in [0,1]$ ▷ ϵ-greedy strategy
    if $(\omega < \epsilon)$ **then**
      Choose action a_t randomly
    else
      Choose action $a_t = \max_a Q_\theta(s_t,a)$
    end if
    Execute action a_t and observe reward r_t and state s_{t+1}
    Form target $y_t = r_t + \max_a Q_\theta(s_{t+1},a)$
    Perform gradient descent using $(y_t - Q_\theta(s_t,a_t))^2$
    Update θ and s_t
  end for
end for

Still, the basic deep Q-learning algorithm presents several flaws, and many improvements can be added to it:

1. **Experience replay**: To avoid forgetting about its past learning, a replay buffer can be created that contains random previous experiences. This buffer is regularly fed to the network as learning material, so previously acquired behaviors are not erased by new ones. This improvement also reduces the correlation between experiences: as the replay buffer is randomly shuffled, the network experiences past (s_t, a_t, r_t, s_{t+1}) tuples in a different order, and is therefore less prone to learn correlation between these [30,56];
2. **Fixed Q-targets**: When computing the loss (4), the same network is used to evaluate the target and $Q_\theta(s_t,a_t)$. Hence, at every learning step, both the Q-value and the target move, *i.e.* the gradient descent algorithm is chasing a moving target, implying big oscillations in the training. To overcome this, a separate network is used to evaluate the target (target network), and its weight parameters θ^- are updated less frequently than that of the Q-network. This way, the target remains fixed for several Q-network updates, making the learning easier;
3. **Double DQN**: The risk of using a single Q-network is that it may over-estimate the Q-values, leading to sub-optimal policies, as π_θ is derived by systematically choosing the actions of highest Q-values. In [57], two networks are used jointly: a DQN is used to select the best action, while the other one is used to estimate the target value;
4. **Prioritized replay buffer**: In the exploration of the environment by the agent, some experiences might occur rarely while being of high importance. As the memory replay batch is sampled uniformly, these experiences may never be selected. In [46], the authors propose to sample the replay buffer according to the training loss value, in order to fill the buffer in priority with experiences where the Q-value evaluation was poor, meaning that the network will have better training on "unexpected events". To avoid overfitting on these events, the filling of the prioritized buffer follows a stochastic rule.

2.3.2. Deep policy gradient

When the policy π_θ is represented by a deep neural network with parameters θ, the evaluation of $\nabla_\theta J(\theta)$ can be delegated to the back-propagation algorithm (see Section 2.2.1), as long as the gradient of the loss function equals the policy gradient (2). The loss can be expressed as:

$$L(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \log\left(\pi_\theta(a_t|s_t)\right) R(\tau)\right].$$

However, it can be shown that using the full history of discounted reward along trajectory τ is not necessary, as a given action a_t has no influence on the rewards obtained at previous time-stations. Several expressions are possible to use instead of $R(\tau)$, that do not modify the value of the computed policy gradient, while reducing its variance (and therefore the required amount of trajectories to correctly approximate the expected value) [47]. Among those, the advantage function $A(s,a)$ is commonly chosen for its low variance:

$$A(s,a) = Q(s,a) - V(s).$$

The advantage function $A(s,a)$, represents the improvement obtained in the expected cumulative reward when taking action a in state s, compared to the average of all possible actions taken in state s. In practice, it is not readily available and must be estimated using the value function. In practice, this estimate is provided by a separately learned value function. As a result, the loss function writes as:

$$L(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \log\left(\pi_\theta(a_t|s_t)\right) A^{\pi_\theta}(s_t,a_t)\right].$$

The vanilla deep policy gradient algorithm is presented below:

Algorithm 2 Vanilla deep policy gradient.

Initialize policy parameters θ_0
for $k = 0, M$ **do** ▷ Loop over batches of trajectories
  for $i = 0, m$ **do** ▷ Loop over trajectories in batch k
    Execute policy π_{θ_k}
    Collect advantage estimates $A^{\pi_{\theta_k}}(s_t,a_t)$
  end for
  Estimate $L(\theta_k)$ over current batch of trajectories
  Update policy parameters θ_k using stochastic gradient ascent
end for

2.3.3. Advantage actor-critic

As mentioned in Section 2.1.3, actor-critic methods propose to simultaneously exploit two networks to improve the learning performance. One of them is policy-based, while the other one is value-based:

1. The actor $\pi_\theta(s,a)$, controls the actions taken by the agent;

2. The critic $Q_w(s,a)$, evaluates the quality of the action taken by the actor.

Both these networks are updated in parallel, in a time-difference (TD) fashion (one update at each time station) instead of the usual Monte-Carlo configuration (one update at the end of the episode). In practice, the advantage function $A(s,a)$ is preferred to the Q-value for its lower variability. In order to avoid having to evaluate two value functions, the advantage function is usually approximated, using the reward obtained by the actor after action a_t and the value evaluated by the critic in state s_{t+1}:

$$A_w(s_t,a_t) \sim R(s_t,a_t) + \gamma V_w(s_{t+1}) - V_w(s_t).$$

The actor-critic process then goes as follows:

Algorithm 3 Advantage actor-critic.

Initialize policy parameters θ_0 ▷ Actor
Initialize action-value function parameters w_0 ▷ Critic
for k = 0, M **do** ▷ Loop over episodes
  for t = 0, T **do** ▷ Loop over time stations
    Execute action a_t following π_θ, get reward r_t and state s_{t+1} ▷ Actor
    Evaluate target $R(s_t,a_t) + \gamma V_w(s_{t+1})$ ▷ Critic
    Update w to w' using target ▷ Critic
    Evaluate advantage $A_{w'}(s_t,a_t) = R(s_t,a_t) + \gamma V_{w'}(s_{t+1}) - V_{w'}(s_t)$ ▷ Critic
    Update θ to θ' using $A_{w'}(s_t,a_t)$ ▷ Actor
  end for
end for

In the latter algorithm, the actor and critic parameter updates follow the methods described in sections 2.3.1 and 2.3.2. The advantage actor critic algorithm has been developed in asynchronous (A3C [37]) and synchronous (A2C) versions. The version of the algorithm presented here corresponds to a time-difference A2C, although many possible variants exist, such as the n-step A2C.

2.3.4. Trust-region and proximal policy optimization

Trust region policy optimization (TRPO) was introduced in 2015 as a major improvement of vanilla policy gradient [47]. In this method, the network update exploits a *surrogate advantage* functional:

$$\theta_{k+1} = \arg\max_\theta L(\theta_k,\theta),$$

with

$$L(\theta_k,\theta) = \mathop{\mathbb{E}}_{(s,a)\sim\pi_{\theta_k}}\left[\Pi(s,a,\theta,\theta_k)A^{\pi_{\theta_k}}(s,a)\right],$$

and

$$\Pi(s,a,\theta,\theta_k) = \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}.$$

In latter expression, $L(\theta_k,\theta)$ measures how much better (or worse) the policy π_θ performs compared to the previous policy π_{θ_k}. In order to avoid too large policy updates that could collapse the policy performance, TRPO leverages second-order natural gradient optimization to update parameters within a trust-region of a fixed maximum Kullback-Leibler divergence between old and updated policy distribution. This relatively complex approach was replaced in the PPO method by simply clipping the maximized expression:

$$L(\theta_k,\theta) = \mathop{\mathbb{E}}_{(s,a)\sim\pi_{\theta_k}}\left[\min\left(\Pi(s,a,\theta,\theta_k)A^{\pi_{\theta_k}}(s,a), g\left(\varepsilon, A^{\pi_{\theta_k}}(s,a)\right)\right)\right],$$

where

$$g(\varepsilon,A) = \begin{cases}(1+\varepsilon)A & A \geq 0,\\ (1-\varepsilon)A & A < 0,\end{cases}$$

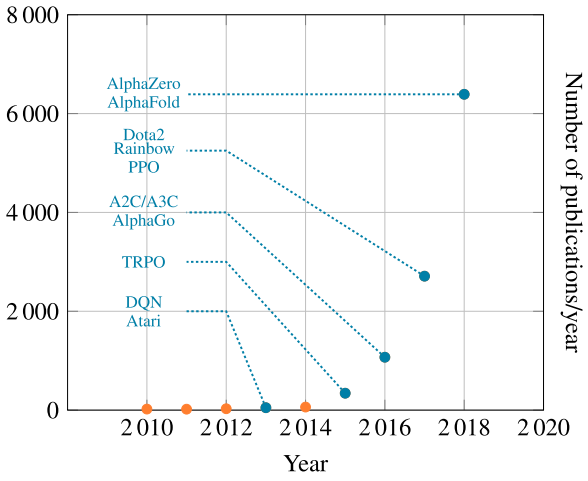


Fig. 3. Number of publications mentioning "Deep reinforcement learning" per year since 2010 (data from www.scholar.google.com). In blue, we indicate some of the most important algorithms or achievements of the field. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

where ε is a small, user-defined parameter. When $A^{\pi_{\theta_k}}(s,a)$ is positive, taking action a in state s is preferable to the average of all actions that could be taken in that state, and it is natural to update the policy to favor this action. Still, if the ratio $\Pi(s,a,\theta,\theta_k)$ is very large, stepping too far from the previous policy π_{θ_k} could damage performance. For that reason, $\Pi(s,a,\theta,\theta_k)$ is clipped to $1+\varepsilon$ to avoid too large updates of the policy. If $A^{\pi_{\theta_k}}(s,a)$ is negative, taking action a in state s represents a poorer choice than the average of all actions that could be taken in that state, and it is natural to update the policy to decrease the probability of taking this action. In the same fashion, $\Pi(s,a,\theta,\theta_k)$ is clipped to $1-\varepsilon$ if it happens to be lower than that value.

Many refinements can be added to these methods that are out of reach of this introduction, such as the use of generalized advantage estimator [48], the introduction of evolution strategies in the learning process [23], or the combination of DQN with PG methods for continuous action spaces [29].

2.4. A conclusion on DRL

Since the first edition of the book of Sutton [53], RL has become a particularly active field of research. However, the domain really started striving after the paper from Mnih et al. [36] and its groundbreaking results using DQN to play Atari games. As of today, the domain holds countless major achievements, and the ability of DRL to learn to achieve complex tasks is well established. It is now exploited in multiple fields of research, and is featured in an ever-increasing amount of publications, as shown in Fig. 3.

A major advantage of DRL is that it can be used as an agnostic tool for control and optimization tasks, both in continuous and discrete contexts. Its only requirement is a well-defined (s_t,a_t,r_t) interface from the environment, which can consist in a numerical simulation or a real-life experiment. As seen earlier, both DQN and DPG ensure a good exploration of the states space, making them able to unveil new optimal behaviors. Moreover, DRL is also robust to transfer learning, meaning that an agent trained on one problem will train much faster on a similar environment. This represents a great strength, especially for domains like fluid mechanics, where computational time represents a limiting factor (Section 3.7 illustrates this point). Transfer learning also introduces a large contrast between DRL and regular optimization techniques, such as adjoint methods, where the product of optimization in a given case cannot

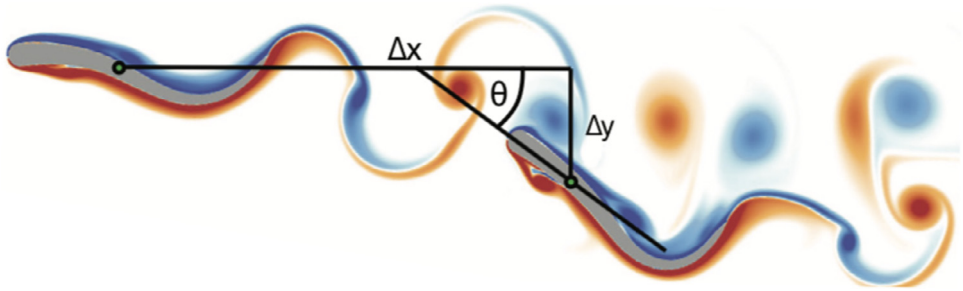


Fig. 4. Leader and follower swimmer, reproduced from [39], with the displacements Δx and Δy as well as the orientation θ between the leader and the follower.

Table 2
Classification of the papers scanned in this review. ADQN and DDPG respectively stand for asynchronous DQN and deep deterministic PG.

Main topic	Algorithm	References
Flow Control	QL	[18]
	DQN	[16,39]
	ADQN	[58]
	TRPO	[32]
	A3C	[15]
	PPO	[43,44]
Homogeneity optimization	QL	[13,19]
	QL	[22,42]
Shape Optimization	QL	[3]
Chaotic Systems	DDPG	[12]

be re-used to speed up optimization in a similar configuration. Finally, as illustrated in Section 3.6, DRL libraries can exploit parallel learning with a close-to-perfect scaling on the available resources. Hence, for its ease of use and its numerous capabilities, DRL represents a promising tool for optimization and design processes involving computational or experimental environments.

3. Applications

In this section, several applications combining DRL and fluid mechanics found in the literature are presented in details. For each case, the numerical experiments and the obtained results are detailed. The choice of DRL algorithm and the problem complexity are also considered. Table 2 presents an overview of the reviewed articles and their corresponding references.

3.1. Synchronised swimming of two fish - [39]

In [39], the authors study the swimming kinematics of two fish in a viscous incompressible flow. The first fish is the leader, and is affected a prescribed gait, while the second fish is a follower, which dynamics are unknown. An objective of the paper is to exploit DRL to derive a swimming strategy that reduces the energy expenditure of the follower. A two-dimensional Navier-Stokes equation is solved to simulate the viscous incompressible flow using a remeshed vortex method. To represent the fish and its undulation, a simplified physical model is used that describes the body curvature $k(s,t)$ of the fish:

$$k(s,t) = A(s)\sin\left[2\pi\left(\frac{t}{T_p} - \frac{s}{L}\right) + \phi\right], \tag{5}$$

where L is the length of the fish, T_p is the tail-beat frequency, ϕ is a phase-difference, t is the time variable, and s is the abscissa. The observation of the environment provided to the DRL agent is composed of:

1. The displacements Δx and Δy (cf Fig. 4),
2. The orientation θ between the follower and the leader (cf Fig. 4),
3. The moment where the action is going to take place (during one period of a tail-beat undulation),
4. The last two actions taken by the agent.

The agent provides actions that correspond to an additive curvature term for expression (5):

$$k'(s,t) = A(s)M\left(\frac{t}{T_p} - \frac{s}{L}\right).$$

The reward to maximize is defined as $r_t = 1 - 2\frac{|\Delta y|}{L}$, which penalizes the follower when it laterally strays away from the path of the leader. Finally, the state space is artificially restricted by terminating the current episode with an arbitrary reward $r_t = -1$ when the two fish get too far away from each other. In this application, a DQN algorithm is used to control the follower. For the best strategy found by the agent, swimming efficiency was increased by 20% compared to baseline of multiple solitary swimmers.

3.2. Efficient collective swimming by harnessing vortices through deep reinforcement learning [58]

The authors in [58] further investigated the swimming strategies of multiple fishes. They work with two and three-dimensional flows (see Fig. 5), and analyze with many details the movement of trained fishes. The DRL model is identical to what is used in [39], but the neural network used was improved by using recurrent neural networks with Long-Short Term Memory (LSTM) layers [21]. The authors underline the importance of this feature, arguing that here, past observations hold informations that are relevant for future transitions (*i.e.* the process is not Markovian anymore). Two different rewards are used: the first one is identical to that of last section, while the second one is simply equal to the swimming efficiency of the follower (see [58]). The result obtained show that collective energy-savings could be achieved with an appropriate use of the wake generated by other swimmers, when compared to the case of multiple solitary swimmers.

3.3. Fluid directed rigid body control using deep reinforcement learning [32]

The control of complex environments that include two fluid flows interacting with multiple rigid bodies (cf Fig. 6) was first proposed by Ma et al. [32]. In this contribution, a multiphase Navier-Stokes solver is used to simulate the full characteristics of the interactions between fluids and rigid bodies at different scales. To provide a rich set of observations to the agent while limiting its input dimension, a convolutional autoencoder is used to extract low-dimensional features from the high-dimensional velocity field of the fluid [31]. This technique allows the use of a smaller neural network with faster convergence. The autoencoder is trained using

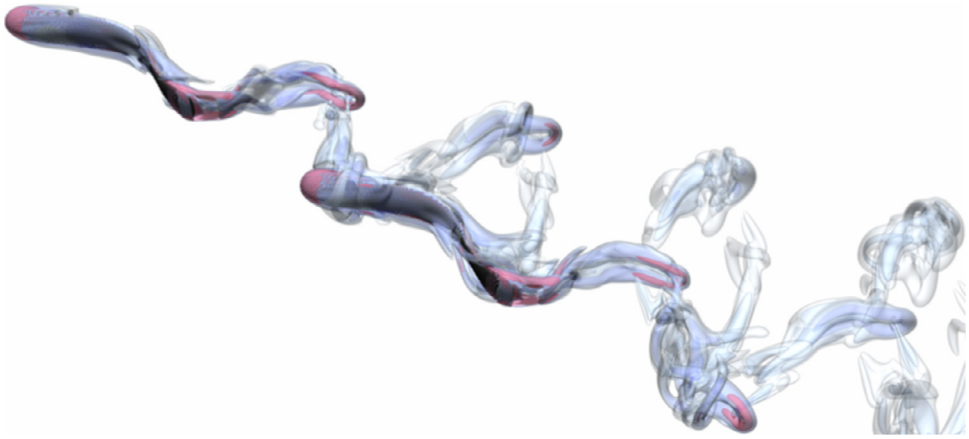


Fig. 5. Leader and follower swimmer, reproduced from Verma et al. [58].

Fig. 6. Fluid jets to control rigid body, reproduced from Ma et al. [32].

a TRPO agent, which received two different observations: (i) extracted features from the fluid velocity field, and (ii) features from the rigid body.

In the environment, the agent is given control over several fluid jets (see Fig. 6), each of them described by 3 parameters: (i) the jet lateral acceleration $\ddot{x}_{jet}$, (ii) the jet angular acceleration $\ddot{\beta}_{jet}$, and (iii) the use of the jet δ_{jet}. Using lateral and angular accelerations instead of lateral and angular positions allows the jet to move smoothly even with a noisy policy. Different rewards were designed depending on the environment goals. A possible example consists in having a rigid body balanced at a fixed position. In this case the designed reward is:

$$r_t = w_c \exp(-||c - c^*||^2) + w_v \exp(-||v||^2) + w_e(1 - \delta_{jet}), \qquad (6)$$

where the first term penalizes a position mismatch according to the goal c^*, the second penalizes the presence of a residual velocity, and the last one encourages the agent to use the smallest amount of control forces. w_c, w_v and w_e are user-defined parameters used to weight the three contributions.

Overall, impressive results were obtained on complex experiments, including playing music with the system or keeping a rotating object at a given position. An important feature of this contribution is the use of the autoencoder to extract features from the high-dimensional velocity fluid fields. Of particular interest is the simultaneous training of the two networks (autoencoder and policy network), leading to a high convergence rate of the overall method.

3.4. Flow shape design for microfluidic devices using deep reinforcement learning [28]

In [28], the authors explore the capabilities of DRL for microfluidic flow sculpting using pillar-shaped obstacles to deform an input flow. Given a target flow, a double DQN is used to generate the corresponding optimal sequence of pillars. The observations provided to the agent consist in the current flow shape, represented as a matrix of black and white pixels. In return, the actions correspond to pillar configurations: the agent can add up to 32 pillars in order to reshape the input flow. A pixel match rate (PMR) function is used to evaluate the similarity between the target flow and the current flow. The reward is then defined using that PMR as:

$$r_t = -(1 - \frac{\text{PMR} - b}{b}),$$

where b is the PMR of the worst-case scenario performed. PMR over 90% were obtained for different target flows, showing that DRL represents a viable alternative to other optimization processes such as genetic algorithms (see Fig. 7). Transfer learning is also exploited, showing that an agent trained on a first flow could be retrained much more efficiently on a different flow than an untrained one, thus saving a lot of computational time.

3.5. Artificial neural networks trained through deep reinforcement learning discover control strategies for active flow control [43]

In [43], the authors present the first application of DRL for performing active flow control in a simple CFD simulation. For this, a simple benchmark is used. More specifically, the flow configuration is based on the one presented in [45] and features a cylinder immersed in a 2D incompressible flow injected following a parabolic velocity profile. The Reynolds number based on the cylinder diameter is moderate to keep the CFD affordable, and following [45], Re=100 is used. The incremental pressure correction scheme is used together with a finite element approach to solve the discretized problem. A periodic Karman vortex street is then obtained, as visible in Fig. 8. Besides, small jets are added on the sides of the cylinder in order to allow controlling the separation of the wake. Therefore, the action performed by the DRL agent is to decide the instantaneous mass flow rate of the small jets, while the reward is chosen to encourage drag reduction. Finally, probes reporting the characteristics of the flow at several fixed points in the computational domain (either pressure or velocity can be used, both providing equivalent results) are added in the vicinity of the cylinder and are used as the observation provided as input to the network.

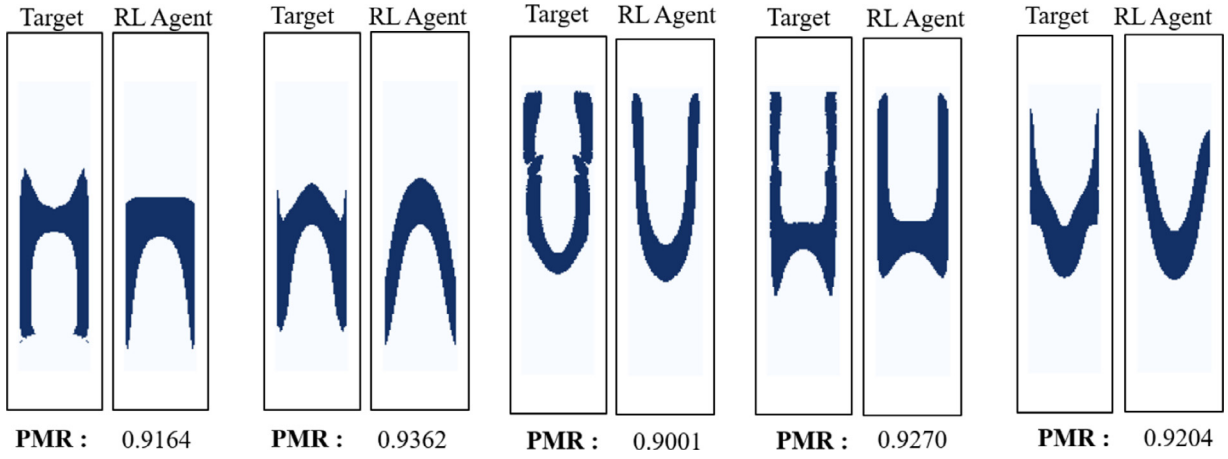


Fig. 7. Targets flow and results obtained from DRL agent, reproduced from Lee et al. [28].

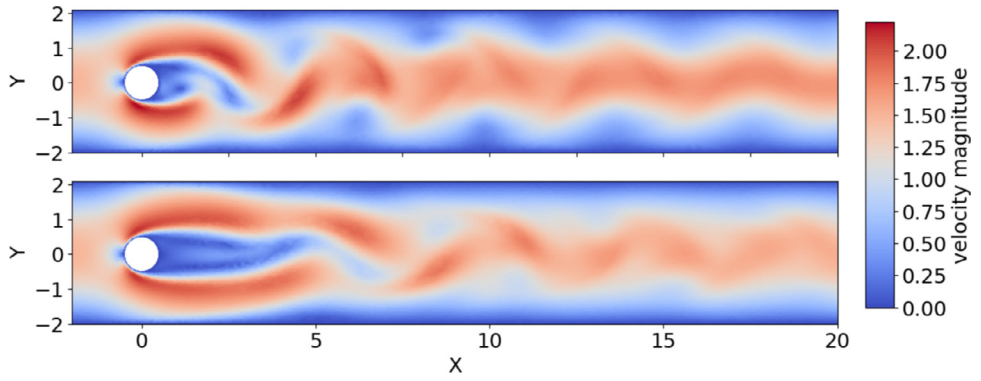


Fig. 8. Comparison of the velocity magnitude without (top) and with (bottom) active flow control, reproduced from Rabault et al. [43]. A clear modification of the cylinder wake, similar to what would be obtained with boat-tailing, is visible.

Good results are obtained, with training performed in around 1300 vortex shedding periods. The control strategy is more complicated than traditional harmonic forcing that had been performed in previous works [7], which illustrates the value of using an ANN as the controller. Typically, around 93% of the drag induced by the vortex shedding effect [7] is suppressed by the control law found, and the strength of the jets needed to reduce the drag is minimal. In the established regime, the magnitude of the mass flow rate injected by the jets normalized by the mass flow rate of the base flow intersecting the cylinder is only of typically 0.6%. The resulting flow, visible in Fig. 8, is similar to what would be obtained with boat tailing; more specifically, an apparent increase in the recirculation bubble is observed.

The work of [43] shows that active flow control is, at least on a simple flow configuration, achievable with small actuations if an adapted control law is used. While there are still many unanswered questions at the moment, such as the robustness of the control law, the application of the methodology to higher, more realistic (from an industrial point of view) Reynolds numbers, or the possibility to control 3D flows, this work establishes DRL as a new methodology that should be further investigated for the task of performing active flow control.

3.6. Accelerating deep reinforcement learning of active flow control strategies through a multi environment approach [44]

One of the main practical limitations of [43] lies within the time needed to perform the learning - around 24 h on a modern CPU. This comes from the inherent cost of the CFD, and the fact that it is challenging to obtain computational speedups on small tasks. In particular, increasing the number of cores used for solving the CFD problem of [43] provides very little speedup, as the communication between the cores negates other speed gains. While this is particularly true on such a small computational problem, any CFD simulation will only speed up to a maximum point independently of the number of cores used. Therefore, a solution is needed to provide the data necessary for the ANN to perform learning in a reasonable amount of time.

In [44], the collection of data used for filling the memory replay buffer is parallelized by using an ensemble of simulations all running in parallel, independently of each other. In practice, this means that the knowledge obtained from several CFD simulations running in an embarrassingly parallel fashion can be focused on the same ANN to speed up its learning.

This technique results in large speedups for the learning process, as visible in Fig. 9. In particular, for a number of simulations that are a divider of the learning frequency of the ANN, perfect scaling is obtained, and the training is formally equivalent to the one obtained in serial (*i.e.* , with one single simulation). However, even more simulations can be used, and it is found empirically that the resulting off-policy data sampling does not sensibly reduce the learning ability, as visible in Fig. 9. This means in practice that the authors achieve a training speedup factor of up to around 60.

This natural parallelism of the learning process of ANNs trained through DRL represents a significant advantage for this methodology, and a promising technical result that opens the way to the control of much more sophisticated situations. It is worth noting

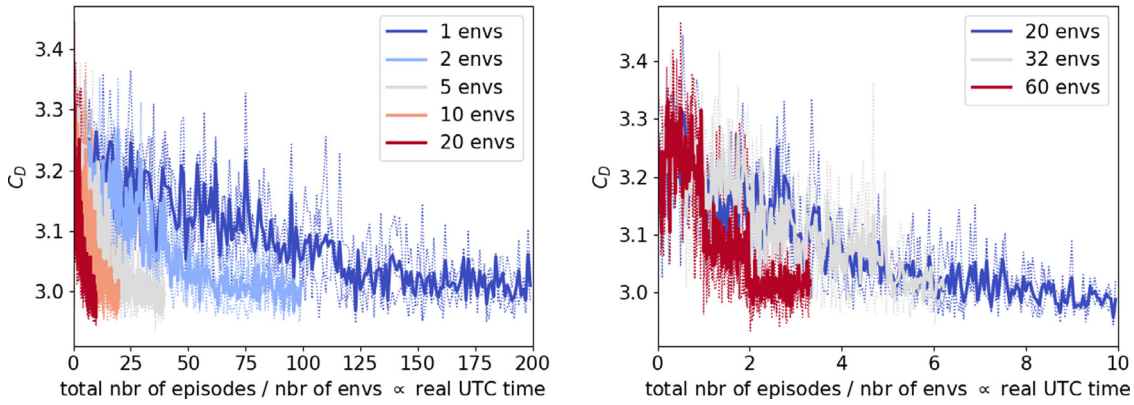


Fig. 9. Illustration of the acceleration of ANN training through DRL using a multi environment approach (reproduced from Rabault and Kuhnle [44]). Both plots illustrate the convergence of the ANN learning as a function of UTC time, depending on the number of environments used in parallel. In both cases, 3 repetitions are performed for each number of environments. Thin lines indicate individual learnings. The average of all three learnings in each case is indicated by a thick line. In the case when the number of environments is a divider of the learning frequency of the ANN, the learning is formally equivalent to the serial case and perfect scaling is observed (left). If a larger number of environments is used, some stepping is observed in the learning process, but this does not affect the ability to perform learning, nor the general speedup of the process (right).

Table 3
Summary of the six trained DRL agents from Garnier and Viquerat [15].

Agent	*Re*	Objective	Details
1.1	10	Control over 17 time steps	Trained from sratch
1.2	10	Direct optimization	Trained from sratch
2.1	40	Control over 17 time steps	Transfer learning from 1.1
2.2	40	Direct optimization	Transfer learning from 1.2
3.1	100	Control over 17 time steps	Transfer learning from 1.1
3.2	100	Direct optimization	Transfer learning from 1.2

that even larger speedups can be obtained by making use of surrogate or reduced-order models, either pre-trained or trained on-the-fly, although specific care must be taken regarding the validity range of these models (as the agent could end up providing inputs to the surrogate model that would not match its training range).

3.7. Sensitivity of aerodynamic forces in laminar flows past a square cylinder

In this section, we propose an original comparison between a classical optimization approach and DRL. The test case is inspired by Meliga et al. [35], where two cylinders are immersed in a moderate Reynolds flow. One is a square main cylinder, which position is fixed, while the other is a cylindrical control cylinder, which position can vary, and which radius is considered much smaller than that of the main cylinder. The goal of this application is to optimize the position of the small cylinder, such that the total drag of the two cylinders is inferior to that of the main cylinder alone. An incompressible Navier-Stokes solver [2] is used to simulate the flow at different Reynolds number past the square cylinder. Moreover, the flow conditions are identical to that described in [43]. Six different agents are trained in different conditions, as detailed in Table 3.

Two different architectures are proposed. The first one is a classical DRL architecture, similar to what is shown in Fig. 10, but with a difference that episodes were reduced to a single action. In that case, the agent directly attempts to propose an optimal (x, y) position from the same initial state, using a PPO algorithm. On the contrary, in the second architecture, successive displacements $(\Delta x, \Delta y)$ of the control cylinder are provided by an A3C agent, starting from a random initial position. The observations provided to the agent are made of the last two positions of the control cylinder, the Reynolds number and encoded features from the velocity fields. To that end, an autoencoder was trained simultaneously to the agent to reduce the state dimensionality (with a dimension reduction from around 8000 to 70), in a similar fashion to [32] (see Fig. 10). The same reward function is used for both architectures:

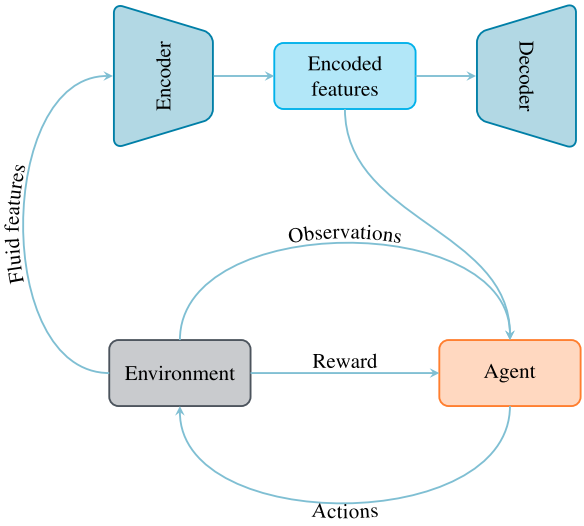
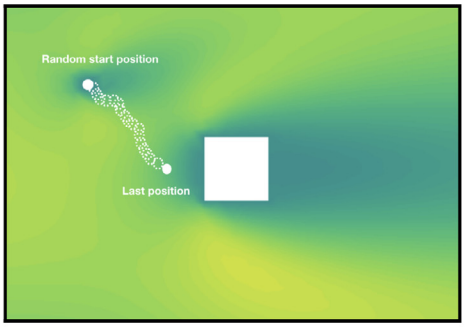


Fig. 10. Architecture for agents 1.1, 2.1 and 3.1.

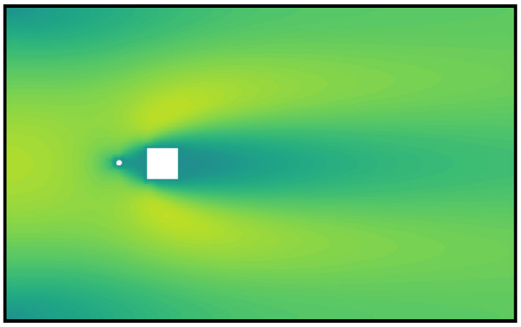
$$r_t = C_D^0 - C_D,$$

where C_D^0 is the drag of the main cylinder alone, C_D is the total drag of the main and the control cylinders, and the drag coefficients are counted positively. With this simple expression, a positive reward is obtained when the combined drag of the cylinders is inferior to the reference drag value.

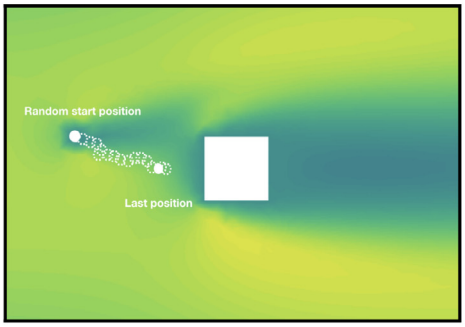
We compare the results obtained with those from Meliga et al. [35], where a classical adjoint method is used. Overall, the same optimal positions were found for the control cylinder, at $Re = 40$ and $Re = 100$. Although the case $Re = 10$ was not present in the original paper, coherent results were obtained for the configuration too. For the first agents at $Re = 10$, training took approximately 5 h using parallel learning with 17 CPUs. Regarding the last four agents, the transfer learning allowed them to learn at a much



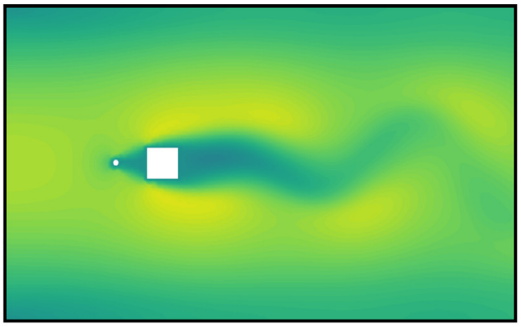
(a) Small displacements, $Re = 10$ (agent 1.1)



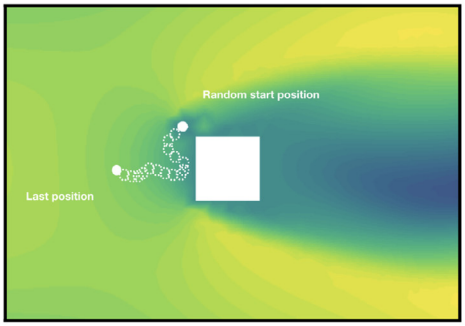
(b) Direct optimization, $Re = 10$ (agent 1.2)



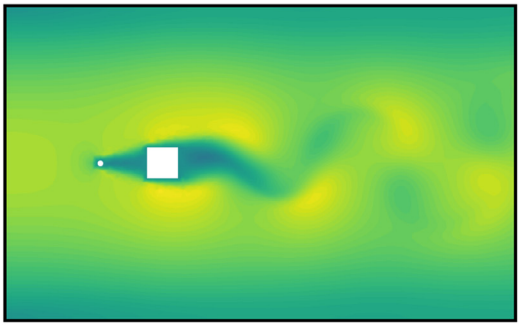
(c) Small displacements, $Re = 40$ (agent 2.1)



(d) Direct optimization, $Re = 40$ (agent 2.2)



(e) Small displacements, $Re = 100$ (agent 3.1)



(f) Direct optimization, $Re = 100$ (agent 3.2)

Fig. 11. Results obtained by the trained agents. The environment consists of a square of unit lateral size centered in (0,0), immersed in a rectangular domain of dimension $[-5, 20] \times [-5, 5]$.

faster pace: only 10 h were necessary on a single CPU. An overview of the results is shown in Fig. 11.

With this experiment, we showed that not only DRL could perform both control and direct optimization tasks, but also that transfer learning between different configurations represents a powerful feature to save computational time. We also confirmed that the use of an autoencoder is a safe and robust way to extract essential features from complex and high dimensional fluid fields. Details about this test case and the code used to reproduce these results can be found on the GitHub repository: https://github.com/DonsetPG/fenics-DRL. We recall that this code uses different library:

1. FEniCS [2] for the CFD solver,
2. Gym [10] for the DRL environment,
3. Stable-baselines [20] for the DRL algorithms.

Future work on this library will focus on generalizing the coupling of DRL with different open source CFD code.

4. Conclusion

Although DRL has already been applied to several cases of optimization and control in the context of fluid dynamics, the literature on the topic remains particularly shallow. In the present article, we reviewed the available contributions on the topic to provide the reader with a comprehensive state of play of the possibilities of DRL in fluid mechanics. In each of them, details were provided on the numerical context and the problem complexity. The choices of the DRL algorithm and the reward shaping were also described.

Given the high-level interfaces of existing libraries, the coupling of DRL algorithms with existing numerical CFD solvers can be achieved with a minimal investment, while opening a wide range of possibilities in terms of optimization and control tasks. The algorithms presented in this review proved to be robust when exposed to possible numerical noise, although high *Re* applications remain to be achieved. Additionally, the parallel capabilities of the main DRL libraries represent a major asset in the context of time-

expensive CFD computations. Transfer learning also proved to be a key feature in saving computational time, as re-training agents already exploited in similar situations led to a fast convergence of the agent policy. The use of autoencoders to feed the agent with both a compact and rich observation of the environment also showed to be beneficial, as it allows for a reduced size of the agent network, thus implying a faster convergence.

As of now, the capabilities and robustness of DRL algorithms in highly turbulent and non-linear flows remain to be explored. Also, their behavior and convergence speed in action spaces of high dimensionalities is unknown. It makes no doubt that the upcoming years will see the mastering of these obstacles, supported by the constant progress made in the DRL field and driven by the numerous industrial challenges that could benefit from it.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Paul Garnier: Investigation, Writing - review & editing. **Jonathan Viquerat:** Methodology, Software, Writing - review & editing. **Jean Rabault:** Visualization, Investigation, Writing - review & editing. **Aurélien Larcher:** Visualization, Investigation, Writing - review & editing. **Alexander Kuhnle:** Software, Validation, Writing - review & editing. **Elie Hachem:** Conceptualization, Software, Validation, Writing - review & editing, Supervision.

References

[1] Ali N, Behdinan K. Optimal geometrical design of aircraft using genetic algorithms. Trans Can Soc MechEng 2003;26:373–88.
[2] Alnæs MS, Bletcha J, Hake A, Johansson B, Kehlet B, Logg A, et al. The FEniCS project version 1.5. Arch Numer Softw 2015;3.
[3] Aeronautics A., editor. Morphing airfoils with four morphing parameters. 2008.
[4] Bahdanau D, Brakel P, Xu K, Goyal A, Lowe R, Pineau J, et al. An actor-critic algorithm for sequence prediction. CoRR 2016(2015):1–17. arXiv:1607.07086.
[5] Bellman R. A Markovian decision process. J Math Mech 1957;6(5):679–84.
[6] Bellman R, Dreyfus SE. Applied dynamic programming. NJ: Princeton University Press Princeton; 1962.
[7] Bergmann M, Cordier L, Brancher J-P. Optimal rotary control of the cylinder wake using proper orthogonal decomposition reduced-order model. Phys Fluids 2005;17(9):097101.
[8] Bewley A, Rigley J, Liu Y, Hawke J, Shen R, Lam V-D, et al. Learning to drive from simulation without real world labels. arXiv e-prints 2018. arXiv:1812.03823.
[9] Bottou L, Curtis FE, Nocedal J. Optimization methods for large-Scale machine learning. SIAM Rev 2018;60:223–311. arXiv:1710.06542.
[10] Brockman G, Cheung V, Pettersson L, Schneider J, Schulman J, Tang J, et al. OpenAI gym. arXiv e-prints 2016.
[11] Brown N, Sandholm T. Superhuman AI for multiplayer poker. Science 2019. doi:10.1126/science.aay2400.
[12] Bucci MA, Semeraro O, Allauzen A, Wisniewski G, Cordier L, Mathelin L. Control of chaotic systems by deep reinforcement learning. arXiv 2019. arXiv:1906.07672.
[13] Colabrese S, Gustavsson K, Celani A, Biferale L. Flow navigation by smart microswimmers via reinforcement learning. Phys Rev Lett 2017;118(15):158004.
[14] François-lavet V, Henderson P, Islam R, Bellemare MG. An introduction to deep reinforcement learning. Foundations and trends in machine learning 2018.
[15] Garnier P., Viquerat J.. Position of a control cylinder to reduce drag. https://github.com/DonsetPG/fenics-DRL; 2019.
[16] Gazzola M, Tchieu AA, Alexeev D, de Brauer A, Koumoutsakos P. Learning to school in the presence of hydrodynamic interactions. J Fluid Mech 2016;789:726–49.
[17] Goodfellow I, Bengio Y, Courville A. The deep learning book. MIT Press; 2017.
[18] Guéniat F, Mathelin L, Hussaini MY. A statistical learning strategy for closed-loop control of fluid flows. Theor Comput Fluid Dyn 2016;30(6):497–510.
[19] Gustavsson K, Biferale L, Celani A, Colabrese S. Finding efficient swimming strategies in a three dimensional chaotic flow by reinforcement learning. arXiv e-prints 2017. arXiv:1711.05826.
[20] Hill A., Raffin A., Ernestus M., Gleave A., Traore R., Dhariwal P., Hesse C., Klimov O., Nichol A., Plappert M., Radford A., Schulman J., Sidor S., Wu Y.. Stable baselines. https://github.com/hill-a/stable-baselines; 2018.
[21] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput 1997;9:1735–80.
[22] Hou Tsang AC, Tong PW, Nallan S, Pak OS. Self-learning how to swim at low Reynolds number. arXiv 2018. arXiv:1808.07639.
[23] Houthooft R, Chen RY, Isola P, Stadie BC, Wolski F, Ho J, et al. Evolved policy gradients. arXiv e-prints 2018. arXiv:1802.04821.
[24] Howard RA. Dynamic programming and Markov processes. New York: Technology Press and Wiley; 1960.
[25] Kendall A, Hawke J, Janz D, Mazur P, Reda D, Allen J-M, et al. Learning to drive in a day. arXiv e-prints 2018. arXiv:1807.00412.
[26] Knight W.. Google just gave control over data center cooling to an AI. https://www.technologyreview.com/s/611902/google-just-gave-control-over-data-center-cooling-to-an-ai/; 2018.
[27] Lee J, Hajela P. Parallel genetic algorithm implementation in multidisciplinary rotor blade design. J Aircr 1996;33:962–9.
[28] Lee XY, Balu A, Stoecklein D, Ganapathysubramanian B, Sarkar S. Flow shape design for microfluidic devices using deep reinforcement learning. CoRR 2018:1–10. arXiv:1811.12444.
[29] Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, Silver D, Wierstra D. Continuous control with deep reinforcement learning. arXiv e-prints 2015. arXiv:1509.02971.
[30] Lin L-J. Reinforcement learning for robots using neural networks, Pittsburgh, PA, USA: Carnegie Mellon University; 1993. Ph.D. thesis. UMI Order No. GAX93-22750
[31] Liou C-Y, Cheng W-C, Liou J-W, Liou D-R. Autoencoder for words. Neurocomputing 2014;139:84–96.
[32] Ma P, Tian Y, Pan Z, Ren B, Manocha D. Fluid directed rigid body control using deep reinforcement learning. ACM Trans Graph 2018;37:1–11.
[33] Mäkinen RAE, Periaux J, Toivanen J. Multidisciplinary shape optimization in aerodynamics and electromagnetics using genetic algorithms. Int J Numer Methods Fluids 1999;30:149–59.
[34] Matos R, Laursen T, Vargas J, Bejan A. Three-dimensional optimization of staggered finned circular and elliptic tubes in forced convection. Int J Therm Sci 2004;43:477–87.
[35] Meliga P, Boujo E, Pujals G, Gallaire Fc. Sensitivity of aerodynamic forces in laminar and turbulent flow past a square cylinder. Phys Fluids 2014;26:26,104101. doi:10.1063/1.4896941. https://hal.archives-ouvertes.fr/hal-01082600
[36] Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, et al. Playing atari with deep reinforcement learning. CoRR 2013. arXiv:1312.5602.
[37] Mnih V, Puigdomènech Badia A, Mirza M, Graves A, Lillicrap TP, Harley T, et al. Asynchronous methods for deep reinforcement learning. arXiv e-prints 2016. arXiv:1602.01783.
[38] Muyl F, Dumas L, Herbert V. Hybrid method for aerodynamic shape optimization in automotive industry. Comput Fluids 2004;33:849–58.
[39] Novati G, Verma S, Alexeev D, Rossinelli D, van Rees WM, Koumoutsakos P. Synchronised swimming of two fish. Bioinspiration Biomimetics 2017;12(3):036001.
[40] OpenAI. OpenAI Five. https://blog.openai.com/openai-five/; 2018.
[41] Pinto L, Andrychowicz M, Welinder P, Zaremba W, Abbeel P. Asymmetric actor critic for image-based robot learning. CoRR 2017. arXiv:1710.06542.
[42] Qiu J, Zhao L, Xu C, Yao Y. Swimming strategy of settling elongated microswimmers by reinforcement learning. arXiv 2018. arXiv:1811.10880.
[43] Rabault J, Kuchta M, Jensen A, Réglade U, Cerardi N. Artificial neural networks trained through deep reinforcement learning discover control strategies for active flow control. J Fluid Mech 2019;865:281–302.
[44] Rabault J, Kuhnle A. Accelerating deep reinforcement learning of active flow control strategies through a multi-environment approach. arXiv e-prints 2019. arXiv:1906.10382.
[45] Schäfer M, Turek S, Durst F, Krause E, Rannacher R. Benchmark computations of laminar flow around a cylinder. In: Flow simulation with high-performance computers II. Springer; 1996. p. 547–66.
[46] Schaul T, Quan J, Antonoglou I, Silver D. Prioritized experience replay. arXiv e-prints 2015. arXiv:1511.05952.
[47] Schulman J, Levine S, Moritz P, Jordan MI, Abbeel P. Trust region policy optimization. arXiv e-prints 2015. arXiv:1502.05477.
[48] Schulman J, Moritz P, Levine S, Jordan M, Abbeel P. High-dimensional continuous control using generalized advantage estimation. arXiv e-prints 2016:1–14. arXiv:1506.02438.
[49] Siegelmann HT, Sontag ED. On the computational power of neural nets. J Comput Syst Sci 1995;50(1):132–50. doi:10.1006/jcss.1995.1013. http://www.sciencedirect.com/science/article/pii/S0022000085710136
[50] Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, Hubert T, Baker L, Lai M, Bolton A, Chen Y, Lillicrap T, Hui F, Sifre L, van den Driessche G, Graepel T, Hassabis D. Mastering the game of Go without human knowledge. Nature 2017;550.
[51] Strang G. Linear algebra and learning from data. Wellesley-Cambridge Press; 2019.
[52] Sutton RS. Learning to predict by the method of temporal differences. Mach Learn 1988;3(1):9–44.
[53] Sutton RS, Barto AG. Reinforcement learning: an introduction. Cambridge, MA: MIT Press; 1998.

[54] Sutton RS, Barto AG. Reinforcement learning: an introduction. Cambridge, MA: MIT Press; 2018.
[55] Tesauro G. Temporal difference learning and TD-gammon. Commun ACM 1995;38.
[56] Tsitsiklis J, Van Roy B. An analysis of temporal-difference learning with function approximation. IEEE Trans 1997:674–90.
[57] van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double q-learning. arXiv e-prints 2015. arXiv:1509.06461.
[58] Verma S, Novati G, Koumoutsakos P. Efficient collective swimming by harnessing vortices through deep reinforcement learning. arXiv e-prints 2018. arXiv:1802.02674.
[59] Vinyals O., Babuschkin I., Chung J., Mathieu M., Jaderberg M., Czarnecki W.M., Dudzik A., Huang A., Georgiev P., Powell R., Ewalds T., Horgan D., Kroiss M., Danihelka I., Agapiou J., Oh J., Dalibard V., Choi D., Sifre L., Sulsky Y., Vezhnevets S., Molloy J., Cai T., Budden D., Paine T., Gulcehre C., Wang Z., Pfaff T., Pohlen T., Wu Y., Yogatama D., Cohen J., McKinney K., Smith O., Schaul T., Lillicrap T., Apps C., Kavukcuoglu K., Hassabis D., Silver D.. AlphaStar: mastering the real-time strategy game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/; 2019.
[60] Williams RJ. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Mach Learn 1992;8(3):229–56.