USING DEEP REINFORCEMENT LEARNING FOR ACTIVE FLOW CONTROL

by

Marius Holm

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

May 2020

Abstract

Acknowledgments

Contents

Chapter 1

Introduction

In this Master thesis we study the application of deep reinforcement learning (DRL) to active flow control (AFC) problems. Until recent years the study of active flow control has been a completely separate field of study compared to the study of neural networks (NNs) and DRL. However, recent advances have opened up a multitude of new approaches to older classical problems, among them the application of machine learning (ML) algorithms to active flow control.

The study of turbulence and how to control it, has been around for thousands of years. Applying feathers to arrows to stabilize the flight path and increase the range of the arrows is one of the earliest examples. In the later years, the study of flows and turbulence have moved more towards numerical simulations which. Simplifying a lot of the experimental expertise that was previously required, to now being able to use pre-made flow solvers and only having to define the system of interest properly. Thus, fluid mechanics (FMs) and AFC have been well researched for a long time, and even more so after the introduction of numerical simulations.

In contrast to the very old study of turbulence and turbulence control we have the second necessary part of this thesis, deep learning and artificial neural networks (ANNs). Artificial neural networks mimic the neural networks found in the human brain, which consists of billions of neurons communicating through electrical signals. ANN research had its first high in the 1940's due to the work done by McCulloch and Pitts [22]. In the 1960's Rosenblatt's perceptron convergence theorem caused another surge of interest, but after Minsky and Papert's work showing the limitations of a single perceptron the interest quickly faded [14]. After almost 20 years of very little activity the introduction of the back-propagation algorithm for the multilayer perceptron model once more resurrected the research into ANNs [14]. Since the 1980's ANN research has been actively pursued and is now also becoming an increasingly important part of industry. A small and simple ANN is not very computationally intensive, but some problems like image- and speech recog-

nition benefit greatly from a larger NN. Thus, solving these kind of problems was for a long time unrealistic due to computational limitations, however we now have powerful enough computers to compute increasingly large and complex ANNs. It is expected that the increase in computing power which has followed Moore's law for approximately 50 years, will soon slow down [42]. As such, making algorithms and NNs more effective will make it possible to continue solving more and more advanced problems without purely relying on the computer chip industry. Han et al. [12] presents one way of reducing the number of parameters needed in a NN without losing accuracy.

1.1 Motivation

ML and artificial intelligence (AI) methodology has seen huge developments in the last few decades which has caused a great variety of new approaches to problems that have previously been unapproachable or very challenging with classical methods. Research of optimization and flow control in fluid mechanics has found optimal control methods for simple linear systems, while for more complex systems stochastic or gradient-based methods have been able to achieve good non-optimal control methods. The work of this project will be in the overlap of machine learning in fluid mechanics and optimization in fluid mechanics, where we apply ML methodology to what is an optimization task.

Combining the fields of machine learning (ML), more specifically deep reinforcement learning (DRL), and active flow control (AFC) is still in the early stages of research where Rabault et al. [33] was among the first to apply DRL algorithms to control a fluid flow. In the paper it was found that a DRL agent was able to significantly reduce the drag in the fluid mechanical system known as the Kármán vortex street behind a single cylinder by controlling a set of jets.

This thesis continues the work presented in Rabault et al. [33] and Rabault and Kuhnle [31] by applying the same methodology to the more complex problem of the fluidic pinball, introduced by Deng et al. [5]. If the same methodology is successful at controlling the fluidic pinball we have shown that the methodology is applicable to more complex control problems, which in turn can inspire further research at increasingly complex problems.

1.2 Thesis structure

In part I we start by giving a thorough introduction to machine learning, then a briefer introduction to deep learning and reinforcement learning. The final chapters of part I are dedicated to introducing flow control, the finite element

method, and giving a brief summary of the two journal articles the project is based upon.

In part II we describe the technical implementations utilized in the project and present some of the key scripts of code for running simulations. The last chapter of part II introduces the simulation environment of the fluidic pinball, including mesh refinement simulations, the creation of initializated flow fields, and active flow control configurations.

Part III presents the results of controlled flow simulations at two different Reynolds numbers (*Re*s) using DRL agents to both reduce and increase drag. We also compare the results of the different agents with simpler control strategies like constant control and sinusoidal control functions.

Part IV includes concluding remarks, a discussion of the results, and examples of future work relevant to the project.

Part I

Theory

Chapter 2

Machine Learning ML

2.1 Introduction

Machine learning is an essential tool used in the work done in this thesis where we apply deep reinforcement learning algorithms to active flow control problems. Deep reinforcement learning is an advanced part of ML, and is not as well known and widely used as regression models, supervised and unsupervised machine learning. In this chapter we want to help the reader obtain a basic understanding of machine learning and present some of the more important building blocks of modern day machine learning (ML). In later chapters we will give more in-depth explanations of the specific algorithms, techniques, and theories which have been directly applied in the project.

We will begin by presenting a selection of classic machine learning tasks, and what a "machine learning algorithm" consists of. As a relatively simple example we will present the linear regression algorithm. We then take a look at the challenges of fitting a model to data, while avoiding what is known as over- and under-fitting. Then we take a look at some of the finer points of ML like hyperparameters and regularization. Finally we will take a brief look at different methods of measuring the success of a machine learning algorithm on a given set of data, and how measuring performance differs between ML algorithms and their use-case.

2.2 Learning algorithms and use cases

A computer algorithm that is able to learn from data is called a machine learning algorithm. Mitchell [25] gives the general definition of a machine learning algorithm as "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Due to the considerable diversity of experiences E, tasks T, and performance measures P, giving a formal definition of each category would be futile in our context. Instead we will give a few examples of each category to give the reader an intuitive introduction the machine learning algorithm. Further details and examples can be found in the books *Deep Learning* by Goodfellow et al. [11] and *Machine Learning* by Mitchell [25].

2.2.1 ML Tasks, T

With machine learning (ML) algorithms we are now able to solve problems which were unattainable with fixed programs written by humans. ML is of special interest to scientists and society because the development of algorithms which are able to *learn* also gives us a peek at what constitues *intelligence* [11].

There's an important distinction to make between the terms "learning" and "task". In our context the algorithm "learning" is our way of solving a given "task" [11]. For example, if we want to make a car that can drive on its own, then the autonomous driving is our task, while "learning" is how the car gains the ability to solve the task of driving without user input. We could try to write our own program manually describing to the car how to drive, but given the massive complexity of combining lights, speed limits, cornering, lane shifting, other cars, animals, and pedestrians such a program would certainly fail sooner or later when approaching an unknown situation. Autonomous cars is one of the most complex problems ML is applied to, and while it is still quite early on in development, it is way ahead of what any programmer would be able to do without a learning algorithm.

A task is often described by a collection of features collected from the object or event we want the ML algorithm to learn from. This collection is often called an *example* and is typically represented by a vector $\mathbf{x} \in \mathbb{R}^n$, where each entry x_i is a feature. Taking one of the most intuitive examples of an image, each feature would be the value of a pixel in the image [11]. Examples of classical machine learning tasks are:

- **Classification:** This type of task is one of the greatest successes by machine learning algorithms. Classification or categorization asks a computer algorithm to determine which of k catogories an input belongs too. Solving a task like this requires the algorithm to produce a function $f : \mathbb{R}^n \to \{1, \ldots, k\}$. When we give the function an input $\mathbf{x}$, the machine learning model produces an output $f(\mathbf{x}) = y$, where the discrete value of y determines which category the input is classified into. Instead of having the algorithm learn a function outputting a single value, the function could learn to produce a probability distribution over the different classes [11]. In the well-known ImageNet challenge,

both variants are used, where you usually present error rates for both top-1 predictions, as well as top-5 predictions [17]. Classification is today best done using deep learning, which revolutionized the ImageNet challenge in 2012 when Kriezhevsky et al. [17] introduced deep learning with convolutional neural networks (CNNs) to the task.

- **Regression:** In regression tasks we want to predict a numerical value based on a given input. Thus, the learning algorithm has to learn a function $f : \mathbb{R}^n \to \mathbb{R}$. Regression problems are quite similar to classification tasks, but the output is usually given in a different format [11]. For example, given input parameters of location, size in square meters, number of rooms, and distance to city center a regression model could output a prediction of the price of a house. Such a prediction value would be of a continuous nature, but with small modifications we could change the task to a classification problem. Instead of outputting a continuous value we could group houses in different price intervals and ask the model to predict which price interval (class) a house with given inputs would fall into.

- **Anomaly detection:** The task of anomaly detection requires the learning algorithm to sift through data and flagging datapoints that do not fit the "normal" behavior of the data. Anomaly detection methods are widely used in fraud detection, insurance, network activity, and fault detection. Detecting fraudulent use of credit cards, faults in infrastructure, or suspicious network activity are some examples of important applications of anomaly detection. Anomalies often lead to actionable information, e.g. credit cards tagged as fraudulent can indicate that the card is stolen or a case of identity theft. Anomaly detection is without a doubt a very important application of learning algorithms, but does not come without challenges. Defining what is normal behavior is very difficult. If large variations in the data is defined as normal we risk missing out on what is actually anomalies, while if we strictly define what's normal we might have too many "anomalies" to actually be able to use the information. Should a bank send notice every time a credit card is used in another city, in another country, or another continent? Anomaly detection models are way more advanced than just looking at the location the card is used in, but works as a simple illustration of the dilemma met when trying to define what is an anomaly or not [3].

There are of course many more tasks possible to solve using learning algorithms, and for a more extensive list we recommend taking a further look at chapter 5.1.1 of Goodfellow et al. [11].

2.2.2 ML Performance Measure, P

After having determined which kind of task we want our learning algorithm to handle, we need a way to measure whether our model is any good or not. This performance measure P is specific to the task T.

Some tasks have quite simple measures of success, such as classification and transcription tasks where we with relative ease can measure the **accuracy** of our model. Where accuracy is just the number of correct outputs divided by the total number of examples. Another, equivalent, measure is the **error rate** of a model, which measures the number of incorrect outputs compared to the total number of examples [11]. For other tasks it would make no sense to use accuracy or error rate as a measure of performance. If the task is to play chess or checkers it would make no sense to talk about accuracy, but keeping track of wins and losses and calculating percent of games won would be a good metric of the models performance [25].

When we are training an algorithm we usually split our data into what is known as a **training set** and a **test set**. The model is then trained solely on the training set and never sees the data in the test set until we determine that the model has finished training. We then use the test set data to evaluate the model on data it has never seen before. If the model performs well on the test set we know the model will most likely perform well in the real world. Should the model however perform poorly on the test set it is likely the model won't work well in the real world either. In section 2.4 we will go into more detail on the use of test sets and how to a create a model that works well, both in training and in the real world.

Choosing a performance metric might seem relatively easy in a lot cases, but choosing a performance measure that actually guides the model towards the desired behavior can be quite challenging. This might be because it's difficult to choose between several options of what should be measured. For example, should we penalize a regression model for making frequent medium-sized mistakes or only penalize rare but large mistakes? For a transcription task we could either measure the accuracy purely on a complete sequence of words, or we could use a finer measure which looks at smaller elements of the sequence [11].

In other cases the difficulties in choosing a performance model might be because the measure we would like to use is not easily extracted from the system. If that is the case we would need to come up with alternate performance measures which are easier to extract, while still giving insight into the models performance [11].

Some tasks have very standardized performance measures, while others are totally up to the imagination of the user. The choice of performance measures can have a huge impact on the learning of a model, as the performance measure will be what tells the model whether it is improving or not. As such

it is of utmost importance to have a clear picture of what the model should do, such that we can come up with a clear and logical measurement to use as feedback for the model.

2.2.3 ML Experience, *E*

Machine learning is usually split into three categories, **supervised**, **unsupervised**, and **reinforcement** learning. This categorization reflects what kind of data, also called experiences, the machine learning algorithm is given. A machine learning algorithm is given a dataset of experiences, each containing multiple features or data points.

Supervised learning is the most commonly used method of machine learning. The dataset provided to the learning algorithm consists of a number of experiences consisting of features relevant to whatever the task is. Importantly each example in the dataset is given a **label**, defining the true output corresponding to each set of datapoints [11]. In an image classification task an experience would be the pixels of an image, and the label would be the correct class, e.g. cat, if the image was of a cat. The algorithm will then study the dataset and learn to identify what images are of cats, dogs, giraffes, elephants, cows, etc. It's important that the dataset be relatively uniform in the distribution of data for each class, such that we don't have 100 images of cats and only 5 images of dogs. Such an imbalance can easily twist the model into "ignoring" dogs and when introduced to new images classify dogs as cats.

Unsupervised learning algorithms are given a dataset only consisting of experiences, without labels, i.e. the algorithm is not trained by teaching what is right and what is wrong. Instead, the algorithm is left to discover underlying structures in the data without any user defined ground-truths. If we are given a very large dataset without any labels, it might be possible to add labels manually, but such an undertaking would take a very long time. In such a case it might be more beneficial to use an unsupervised algorithm, saving the time it would take to add labels, and still being able to draw interesting information from the data. Unsupervised learning is used in applications like clustering where you group a dataset into clusters of similar datapoints, or in tasks like density estimation and denoising where the algorithm should learn the probability distribution of the dataset [11].

The final big category of machine learning is reinforcement learning. In contrast to supervised and unsupervised learning, reinforcement learning algorithms are not given a fixed dataset valid during the entire learning period. Reinforcement learning algorithms interact with an environment, thus the inputs given to the algorithm changes throughout the learning period [11]. A more in-depth explanation of reinforcement learning (RL) will be given in chapter 4.

2.3 Linear Regression

Linear regression is often used as an introductory example to machine learning. Most students have met linear regression problems in high school, although the problems might usually be solved using computer tools. The most straighforward way of solving a linear regression problem requires some simple linear algebra and calculus.

To solve a linear regression problem we want to build a model which can take a vector $\mathbf{x} \in \mathbb{R}^n$ as input and output a value $y \in \mathbb{R}$. Letting $\hat{y}$ be the predicted value of y our model outputs we have:

$$\hat{y} = \boldsymbol{w}^\top \mathbf{x}, \tag{2.1}$$

where $\boldsymbol{w} \in \mathbb{R}^n$ is a vector of **weights**, also called **parameters** [11].

The weights control how the model calculates the predicted output, in this case w_i is the coefficients we multiply with the inputs x_i, before summing the terms from each feature. Thus, if a weight w_i is positive, then increasing the value of the corresponding input x_i will increase the resulting output value $\hat{y}$. Naturally, it follows that increasing the value of an input x_i with a corresponding weight w_i which is negative, will result in the predicted output value to decrease. A large weight means the corresponding feature has a large impact on the predicted output, while a zero weight corresponds to the feature having no effect at all on the prediction [11].

We have thus defined a task T: predict y from the input $\mathbf{x}$ by calculating $\hat{y} = \boldsymbol{w}^\top \mathbf{x}$. In order to know whether out model is performing good or bad we need a performance measure P [11].

By splitting the data we have into two sets, **training** and **test**, we have obtained data that can be used to evaluate the model. We will then train the regression model on the training set, while using the test set to evaluate the models performance. Suppose the test set is a matrix of m inputs, called $\mathbf{X}^{(\text{test})}$, with a corresponding vector of regression targets or "labels", $\boldsymbol{y}^{(\text{test})}$ providing the correct value of y for each example input in $\mathbf{X}^{(\text{test})}$ [11].

We can then measure the performance of the model by computing the mean squared error (MSE) of the model on the test set. Let $\hat{\boldsymbol{y}}^{(\text{test})}$ be the predictions of the model on the test set inputs from $\mathbf{X}^{(\text{test})}$. The MSE is then given by

$$\text{MSE}_{\text{test}} = \frac{1}{m} \sum_i \left(\hat{\boldsymbol{y}}^{(\text{test})} - \boldsymbol{y}^{(\text{test})} \right)_i^2. \tag{2.2}$$

It should be quite obvious that the error decreased to 0 when $\hat{\boldsymbol{y}}^{(\text{test})} = \boldsymbol{y}^{(\text{test})}$. The error measure described is also equivalent to what is known as the Euclidean distance between the predictions and the targets. Usually written as

$$\text{MSE}_{\text{test}} = \frac{1}{m} \left\| \hat{y}^{(\text{test})} - y^{(\text{test})} \right\|_2^2. \tag{2.3}$$

By allowing the algorithm to gain experience by observing the training set $\left(X^{(\text{train})}, y^{(\text{train})}\right)$, the algorithm should find better values for the weights w such that the MSE_{test} is reduced. This can be done by minimizing the MSE on the training set, $\text{MSE}_{\text{train}}$, which can be done by solving for where the gradient is 0 [11].

Before solving the minimization problem of $\text{MSE}_{\text{train}}$ let us introduce a few machine learning terms, namely objective and cost functions. An objective function, $\mathcal{O}$, defines an optimization problem that we either want to maximize or minimize. In machine learning we so often want to minimize the objective function that this got its own name, the cost function C.

Back to the example of linear regression we define $\mathcal{O} = \text{MSE}_{\text{test}}$, and as we want to minimize the MSE, the objective function is also a cost function given by

$$C = \frac{1}{m} \left\| \hat{y}^{(\text{test})} - y^{(\text{test})} \right\|_2^2 \tag{2.4}$$

To simplify notation we will avoid adding "train" as superscript, but note that all the matrices and vectors below are of the training set. Note also that we will use $\hat{y} = Xw$, while keeping the true targets as y.

$$\nabla_w C = 0 \tag{2.5}$$

$$\Rightarrow \nabla_w \frac{1}{m} \left\| \hat{y}^{(\text{train})} - y^{(\text{train})} \right\| = 0 \tag{2.6}$$

$$\Rightarrow \frac{1}{m} \nabla_w \left\| Xw - y \right\| = 0 \tag{2.7}$$

$$\Rightarrow \nabla_w \left(Xw - y \right)^T \left(Xw - y \right) = 0 \tag{2.8}$$

$$\Rightarrow \nabla_w \left(w^T X^T X w - 2 w^T X^T y + y^T y \right) = 0 \tag{2.9}$$

$$\Rightarrow 2 X^T X w - 2 X^T y = 0 \tag{2.10}$$

$$\Rightarrow w = \left(X^T X \right)^{-1} X^T y \tag{2.11}$$

Solving eq. (2.11) is a simple sort of learning algorithm which can be done using several different methods. Methods calculating matrix inversion $\left(X^T X\right)^{-1}$ are possible, but avoiding calculating the matrix inversion has a few benefits like being more computationally efficient and being less prone to errors.

Linear regression often includes an intercept parameter, also called a bias parameter, with which the model can be written as

$$\hat{y} = \boldsymbol{w}^{\mathrm{T}}\mathbf{x} + \mathrm{b}. \tag{2.12}$$

The predictions will still be a linear function, but a non-zero bias parameter will cause the function to not pass through the origin.

Linear regression is a very simple learning algorithm which has very limited use in the real world of complex datasets, but provides a simple introduction to some of the basics behind a learning algorithm. In the next sections we will go more into more specific considerations and methods which are used in more advanced learning algorithms, and that can handle more complex datasets.

2.4 Fitting to data

In machine learning we want to fit a model to the data we have, and while we have access to the raw data, knowing the complexity of the data, and the corresponding need for complexity in our model is not as easy. In addition there is a good chance that the collected data contain a certain amount of noise and measurement errors. A machine learning model is initially trained on the data we have access to, but is usually meant to be used on new data that we don't have yet. Thus, it is important that the model is able to extrapolate what it has learned to new and unseen data. A models ability to perform well on new unseen data is called **generalization**.

When we train our machine learning algorithm we split the data we have into two sets, a training and a test set. When training the model we seek to lower the error on the training set, called the **training error**. If all we wanted to do was reduce the training error we would have simple optimization problem. The difference between optimization and machine learning is that what we ultimately seek is to reduce the **generalization error** or **test error** as much as possible. We use the test set to calculate the test error, as that data is yet unseen by the model and as long as the test set is randomly selected will be a fair representation of unseen data.

In the example of linear regression we trained the model by minimizing the **training error** while what we actually want is to minimize the **test error**.

If the training and test set are completely randomly chosen, minimizing the training error would not necessarily say anything about how the model would perform on the test set. However, by making a few assumptions regarding how the two sets are collected we can say a bit more about how improving the model during training will also improve performance in the test set. The assumptions we have to make are known as **i.i.d. assumptions**. We assume that the samples in the data are **independent** and that the training and test set are **identically distributed**. This means that the same probability

distribution is used when sampling the data to split into training and test sets [11].

After the data has been sampled into a training and a test set, we start training our model on the training set. By updating the parameters of the model we can improve performance on the training set and reduce the training error. When the model is finished training we apply it to the previously unseen data of the test set and observe the test error. Note that because of the assumptions made with regards to collecting the two datasets we know that the expected test error is at best the same as the expected training error, and most often it will be larger than the expected training error [11].

Overfitting and **underfitting** are two very important terms in machine learning literature when we are talking about a models performance. As the names themselves should hint at they describe two issues of opposite nature we might face when working with ML algorithms. Underfitting means that the model is not able to represent the complexity of the data, and the model itself is not complex enough. The result of underfitting will be a larger error value on the training set than what we can accept. Overfitting on the other hand means that our model is too complex and instead of learning the general structures of the data the model "remembers" the training data. This often results in a very small and nice looking training error, but when we introduce the unseen data of the test set the model is not able to generalize and the test error will be quite large compared to the training error. If we are able to avoid under- and overfitting we end up with an "ideal" machine learning model with a low training error and a small gap between the training and test error. Thus, the model is complex enough to represent the data it learns from, while still being generalized enough to perform well on unseen data.

The issues we meet with over- and underfitting are clearly illustrated by looking at a simple polynomial regression problem. We implement the illustration using the python package `numpy` by Van Der Walt et al. [41].

2.5 Regularization

Computing power has been increasing exponentially following Moore's law since the 1970's [42]. As a result we have vast computing resources available which has made increasingly complex machine learning models available to researchers. This has made solving more complex problems possible, while at the same time it has become much easier to encounter the issues with overfitted models. In order to counteract overfitting issues, which in general are more common than underfitting, we have regularization techniques that are used to reduce problems regarding overfitting.

As mentioned in the previous section, an overfitted model performs well on the training set while performing significantly worse on the test set. As

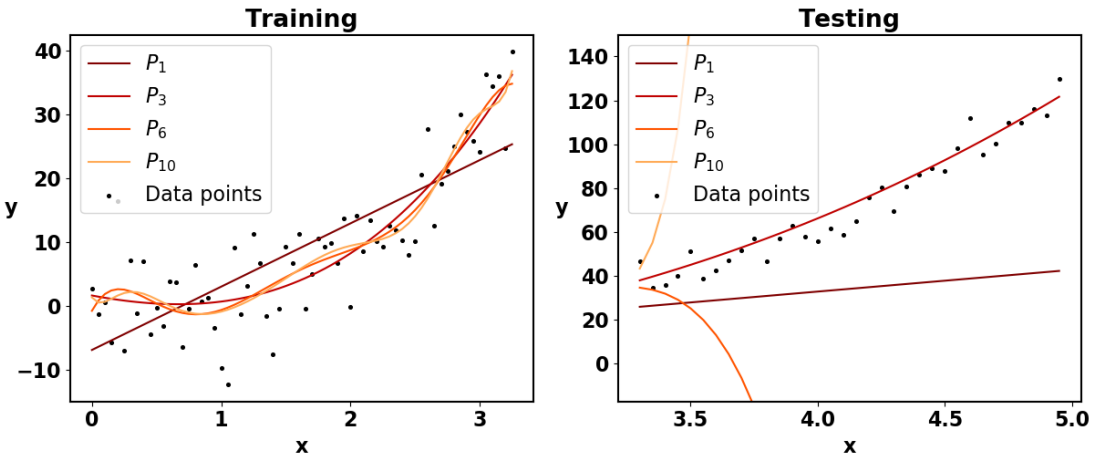


Figure 2.1: Polynomial regression of data produced by a cubic polynomial with added noise drawn from a normal distribution. Polynomial regression models are of varying complexity indicated by the polynomial basis P_n. The models are fitted to the data in the training region, and then evaluated in the testing region. The linear model, P_1, is what we call underfitted. In the training region the linear model performs reasonably well, but clearly worse than the more complex models. When we look at the testing data it becomes clear that a linear model is too simple and is not able to generalize to unseen data. On the other hand we have the very complex models of P_6 and P_{10}. In the training region they seem to follow the data very well, also following minor variations created by the added noise. However, when proceeding to the testing region the more complex models collapse completely and are not able to generalize to the unseen data at all. Thus, the training error of the complex models will be quite small, while the generalization error will be much worse, which we know is very typical of an overfitted model. In the middle we have the optimal solution, the P_3 model, which fits the training data quite well, while at the same time also generalizes very well into the testing region.

such, we don't necessarily want to worsen performance on the training set, as long as we are able to improve the generalization error of the test set. Regularization is therefore a tool we apply to better the generalization error, and not something we use to improve the training error.

Taking the example of linear regression we can include a regularization technique called **weight decay**. In the first example of linear regression we minimized the MSE on the training set. We now add a term to the minimization equation where we take the squared L^2 norm of the weights, which when minimizing the entire equation will push the weights toward smaller values [11]. The equation we minimize will be of the form

$$J(\mathbf{w}) = \text{MSE}_{\text{train}} + \lambda \mathbf{w}^T \mathbf{w}. \tag{2.13}$$

The parameter λ must be chosen ahead of training and determining the magnitude of λ can have significant impact on the results. If $\lambda = 0$ we have no weight decay, and the model will be trained in exactly the same manner as for the standard linear regression problem, and be prone to overfitting to data. If λ is given a large value the model will tend to underfit because the weights are forced so small that the model will tend to have no slope at all. More details and plots of the effect regularization can be found in chapter 5.2.2 of *Deep Learning* by Goodfellow et al. [11].

2.6 Hyperparameters

Hyperparameters are central to machine learning algorithms and are especially important with regards to optimizing and fine-tuning models. In essence, a hyperparameter is a setting used to control the algorithm. Hyperparameters are fixed during training and testing, and are not optimized by the learning algorithm itself. Thus, choosing values for hyperparameters is not done by the learning algorithm itself, although it would be possible to create an additional learning algorithm trained to find the best hyperparameters for the original algorithm.

In the polynomial regression problem presented in section 2.4 we have one hyperparameter, namely the degree of the polynomial fitting. Another example is found in section section 2.5 where the weight decay model for linear regression includes a λ parameter which controls the penalization of large weights.

To help us determine the performance of a chosen set of hyperparameters we introduce another set of data, the **validation set**. This is necessary because we cannot use the test set while still training our model, including choosing hyperparameters. The test set should be isolated until we have fixed the hyperparameters of our choice and are finished training. But we still need some way to determine the proficiency of a given set of hyperparameters, and that is where a validation set comes in handy. Figure fig. 2.2 shows how we split a dataset first into two sets, and then if necessary can split the training set once more in order to optimize hyperparameter configurations. Note that as we are "training" the hyperparameters on the validation set the **validation set error** will underestimate the generalization error of the test set, but usually by less of a margin than the training error.

The two examples previously mentioned, weight decay of linear regression and polynomial regression, are both relatively simple optimization problems as they both only include one hyperparameter. Modern machine learn-

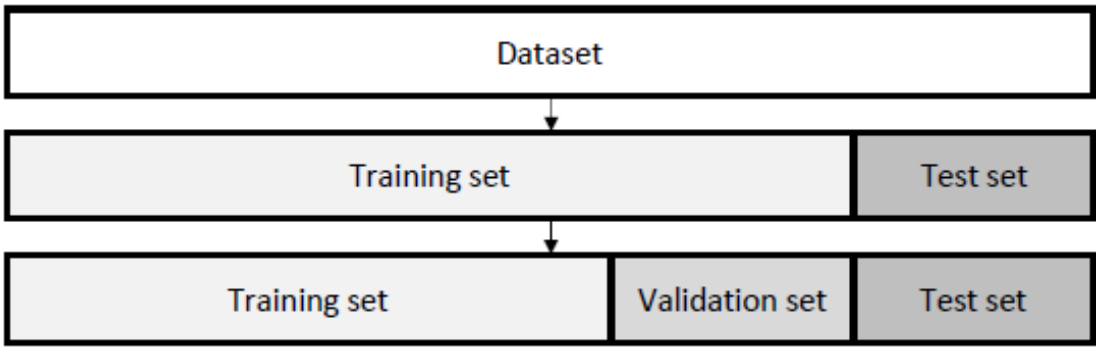


Figure 2.2: The dataset contains all data to be used for a given machine learning task. We first split the data into a training and a test set so we are able to measure the generalization error on the test set. If we also want to optimize our choice of hyperparameters we need a validation set to test the different hyperparameter configurations on. The validation set is created by splitting the initial training set into two parts. The larger part is still called the training set and is used for initial training, while the smaller part is the validation set which we use to test the hyperparameter configurations. When we are happy with the performance of a configuration of hyperparameters on the training and validation sets, we then move on and do a final performance measure on the test set.

ing algorithms are usually dependent on multiple hyperparameters and optimizing a larger set of hyperparameters can be very computationally intensive because we have to train the model for each hyperparameter configuration. There are multiple methods for how one should proceed when optimizing hyperparameters.

The simplest method would be to manually enter hyperparameter values and see how performance changes with changing values. The next method is a more structured approach known as grid search where we set up a multidimensional grid of the parameters and evaluate for every combination on the grid. This method becomes computationally intensive very quickly with increasing numbers of parameters, but usually works better than the manual method. A more efficient method for finding optimal hyperparameters was introduced by Bergstra and Bengio [1] where they show that a simple random search is among the most efficient methods of hyperparameter optimization. Among the reasons for a random search performing so well is the fact that not all hyperparameters have the same importance, and thus a grid search will test a lot of bad configurations with no real improvement. Random search was shown to be more computationally efficient in addition to finding better models in most cases [1].

2.7 Gradient Descent

As previously mentioned optimization is central to machine learning. In section section 2.3 we introduced optimization of an objective function either as a maximization or minimization problem, where minimization is the more common of the two in ML. An **objective function** cast as a minimization problem is often called either a **loss**, **cost** or **error** function.

From calculus we know that a given function has an extremum where the derivative of the function is equal to zero. Simple functions like a third degree polynomials have closed-form solutions of where the derivative is zero and finding minima or maxima of the function are trivial by analytically deriving the function. However for very complex functionals, like a cost function might be, finding closed-form solutions of the first derivative might be either impossible or impractical. In such cases we turn to iterative methods and in the case of machine learning gradient descent methods are very popular. Gradient descent is a first-order iterative optimization method used to find a functions minima. Second-order methods like Newton's method are not widely used in machine learning because of increased computational costs and issues with treating saddle points. Vanilla gradient descent computes the gradient of a cost function J w.r.t. the parameters θ. The parameter update performed by gradient descent is then given by:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta), \tag{2.14}$$

where η determines the size of each update step we take towards a minimum. The gradient vector of the function will point in the direction of the steepest ascent of the function, so in order to move towards a minima we go in the exact opposite direction. The parameter η is known as the **learning rate** and is very important when optimizing ML algorithms. Also note that the learning rate is always chosen to be positive real number, while the magnitude will be dependent on the algorithm and the function to be minimized. A too small value of η will cause convergence to be very slow, while a too large value can cause too large parameter updates causing us to miss the minimum entirely.

The vanilla gradient descent method calculates the gradient of the cost function, $\nabla_\theta J(\theta)$, for the entire training set of data to do just one update. This can make vanilla gradient descent very slow and close to impossible for a dataset that does not fit in memory.

The vanilla gradient descent method, being a first-order method, meets two problems which the gradient descent based alternatives tries to solve. Those two problems being that a cost function will usually have a lot of local minima which can easily cause a vanilla gradient descent method to get stuck, and that the convergence of gradient descent can be very slow or miss a

minima entirely depending on the used configuration, e.g. too large or small learning rate. Although vanilla gradient descent has it's shortcomings we are guaranteed, given enough time, to converge to a global minimum for a convex surface and to a local minimum for a non-convex surface [35].

Because of gradient descent methods being so useful and popular in modern machine learning a multitude of gradient descent variations exist. With added modifications to the base gradient descent method they try to solve the issues mentioned in the previous paragraph. In the next sections we will present a few of the most popular gradient descent based optimization algorithms used in machine learning today.

2.7.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is probably the most well-known modified gradient descent method used in machine learning. Instead of performing an update after computing the gradient of over the entire training set SGD updates the parameters calculating the gradient update over a sampled **mini-batch** of examples drawn uniformly from the training set. The size of such a minibatch can range between 1 and up to a few hundred, and while it might be intuitive to increase the minibatch size as the dataset increases in size this is not necessary. Thus a dataset consisting of billions of examples can be fitted by computing gradient descent updates on small batches of only a few hundred samples [11].

As vanilla gradient descent computes the gradient for each example in the dataset it will recompute gradients for similar examples before updating the parameters. SGD avoids this by only calculating the gradient for each example in the minibatch, which is much less likely to include similar experiences, before updating. SGD with a minibatch size of 1 means that we update the parameters for each example in the training data. The gradient descent equation would then be given as:

$$\theta = \theta - \eta \cdot J\left(\theta; x^{(i)}; y^{(i)}\right), \tag{2.15}$$

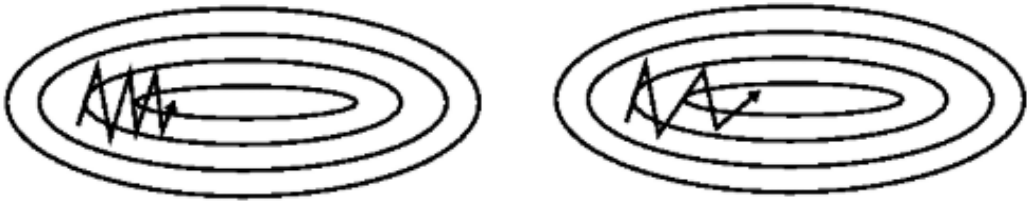
where $x^{(i)}$ is a training example and $y^{(i)}$ is the corresponding label. This makes SGD a lot more computationally effective than vanilla gradient descent and also allows us to add more data after training has been started, so called learning online [35].

Compared to vanilla gradient descent the updates done by SGD will be a lot more fluctuating. This somewhat complicates converging to a specific minimum, but at the same time allows SGD to jump out of a local minima to find a new and potentially better local minima. It has also been shown that by slowly decreasing the learning rate, η, will let SGD reach the same behavior as the vanilla method where we are almost guaranteed to reach a

global minimum for convex functions and a local minimum for a non-convex function.

2.7.2 Momentum Gradient Descent

Stochastic gradient descent (SGD) has trouble dealing with areas of a surface with much larger gradients in one dimension than another, which is common around local minima. This can cause SGD convergence to slow down a lot when closing in on the local optimum. To help in such cases we can introduce **momentum** to the SGD method.



(a) SGD without momentum. **(b)** SGD with momentum.

Figure 2.3: Illustration of SGD oscillations without and with momentum. Source: Genevieve B. Orr (1999), lecture notes CS-449: Neural Networks.

Momentum helps accelerate SGD towards the minimum and at the same time dampen the oscillations of SGD, as illustrated in fig. 2.3. The parameter update for SGD with momentum is written as

$$\begin{aligned} v_t &= \beta\, v_{t-1} + (1-\beta)\, \nabla_\theta J(\theta) \\ \theta_t &= \theta_{t-1} + \eta\, v_t \end{aligned} \tag{2.16}$$

where η is the learning rate. The "velocity" v_t accumulates the previous gradients and the parameter β controls how quickly those contributions should decay. The momentum parameter β is usually given a value of 0.9, but can be modified as long as $\beta \in [0, 1)$. A larger value of β gives more weight to previous gradients, while if $\beta = 0$ we are back to standard SGD.

The addition of momentum can be thought of as rolling a ball down a hill. The momentum of the ball will continue to increase as the velocity increases until air resistance stops acceleration at terminal velocity, where the "air resistance" is given by the parameter β. As the gradient is moving towards a minimum the previous gradients pointing in the same direction will build momentum, while the gradients pointing in other directions will be reduced. Thus oscillations pointing in other directions than the minimum are dampened while momentum drives us quicker towards the minimum.

Note that in some of the literature the equations for SGD with momentum are written differently, e.g. from Ruder [35]:

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta_t &= \theta_{t-1} + v_t. \end{aligned} \tag{2.17}$$

This is a matter of scaling and will impact which values will work best for the learning rate, η, and the decay factor, γ. However, the two methods are equivalent in practice and if the effective values are equivalent the results will be the same.

2.7.3 Adam

Adam was introduced by Kingma and Ba [16] and combines several different gradient descent modifications. Adam is based on stochastic optimization where we use batched data, while also using momentum, and adaptive learning rates. An adaptive learning rate is given as a function of the epoch, the magnitude of the derivative, or both, instead of setting the learning rate as a fixed number.

Note that for vanilla gradient descent and SGD every parameter was updated with the same learning rate at the same time step t. This is not the case for Adam, AdaGrad and RMSProp, and thus we introduce the following notation to make things easier for us:

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i}). \tag{2.18}$$

$g_{t,i}$ is now the gradient of the objective function, J, w.r.t. the parameter θ_i at time step t.

Both RMSProp and AdaGrad compute adaptive learning rates by storing an exponentially decaying average of past squared gradients v_t. Adam does the same, but additionally keeps an exponentially decaying average of past gradients m_t, which works similarly to momentum as introduced in subsection 2.7.2 [35].

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{2.19}$$

m_t and v_t are estimates of the first and second moments of the derivative. The update now contains two β parameters similarly to standard momentum updates. Kingma and Ba [16] notes that because m_t and v_t are initialized as zero-vectors they are biased towards zero, especially when the β parameters are close 1. To avoid such issues Kingma and Ba [16] introduce bias-corrected estimates for the two moments:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t} \end{aligned} \tag{2.20}$$

The final parameter update is then given in the recognizable form:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2.21}$$

Adam was designed to combine the best of AdaGrad by Duchi et al. [6] and RMSProp by Tieleman and Hinton [39], which both were among the more popular optimization methods before Adam. AdaGrad works well with sparse gradients, while RMSProp is great at dealing with non-stationary objectives [16].

For more details on the different versions of gradient descent presented in section 2.7 we recommend the article by Ruder [35] which includes more details and an even wider variety of gradient descent optimizers.

Chapter 3

Deep learning

Deep learning is a sub-field of machine learning where the learning algorithms are based on neural networks (NNs) also called artificial neural networks (ANNs). Neural networks are inspired by neuroscience and the neurons found in the brain, and work as a loose attempt at reproducing the calculations the biological neurons compute. In the brain a neuron activates if it receives strong enough signals from neighboring neurons, and similarly each neuron in a neural network calculates an activation value determining what information is passed along. Note that although neural networks are inspired by the neurons found in the human brain, a modern neural network is guided by a lot of mathematical and engineering disciplines and the main goal is not to model the brain perfectly [11]. As mentioned in chapter 1 neural networks have been popular within research multiple times in the last century, but interest soared after Kriezhevsky et al. [17] with their neural network named AlexNet lowered the ImageNet challenge error rate from 28% to 16%. It has since been an area of widespread interest, not just in scientific research, but this time also in the industry and business world.

In deep learning there are several different kinds of neural networks. In this thesis we use what is known as a dense neural network (DNN) or fully connected neural network (FCNN), which are equivalent. In section 3.1 we will explain more in detail how a neural network is built and explain the mathematics going on behind the scenes in deep learning. We will also touch upon how a NNs are optimized with gradient descent methods like those introduced in section 2.7, which requires a few extra considerations compared to models like linear regression. A neural network is largely described by the layer structure and its activation functions. Different layer types and structures will be discussed in section 3.5, and section 3.3 will introduce a selection of important activation functions found in deep learning. In addition to DNNs there are several other kinds of neural networks like convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Although

we won't implement those types of NNs they are widely used, important to the field of deep learning. Neural networks can take advantage of previously trained models, so-called pre-trained networks, which if applied correctly can reduce training time significantly and also help convergence when moving to more complex problems. Both of these traits were observed in the work of this thesis and an introduction to pre-training neural networks will be given in section 3.6.

3.1 Feedforward neural networks

As mentioned in the introduction to deep learning the term "neural network" describes a rather wide variety of models. In this section we will look at feedforward neural networks (FFNNs), also called deep feedforward networks or multilayer perceptrons (MLPs). The goal of a FFNN is to approximate some function f^*. In the example of a classification task we would want to approximate the function $f^*(\mathbf{x}) = y$, mapping an input $\mathbf{x}$ to a category y. A FFNN would then define an approximate mapping $f(\mathbf{x}; \theta) = \hat{y}$ and learn the values for the parameters θ that results in the best function approximation [11].

"Feedforward" is given to the network because information goes from the input $\mathbf{x}$, through intermediate computations, and then finally evaluates the output $\hat{y}$. If there were feedback connections where outputs of the model is fed back into itself we would have a recurrent neural network (RNN).

A neural network in general consists of computational nodes called **neurons**. The neurons are structured in layers where information is passed forward from one layer to the next until the output is given by the final layer. The layers in a neural network is split into three categories where the first layer is known as the **input layer**. The input layer is usually given as a vector of inputs, e.g. pixel values of an image or a vector of values. Thus, the size of the input layer must match the size of the examples given in the training data. The next layers are called **hidden layers** and the number of hidden layers, as well as the size of each layer, is dependent on the complexity of our task. These layers are named "hidden" as the behavior of the hidden layers are left up to the learning algorithm and are not determined by the training data. The final layer is the **output layer** which outputs the final result of the neural network. The output layer is, like the input layer, limited somewhat by the training data. If the training data is labeled in a binary fashion the output layer must output either a 0 or a 1. However, if the problem is a regression problem the labels and thus outputs might only be specified to be a real number.

Each layer, except the input layer, in a neural network is given an **activation function**. The activation function is often the same for all of the hidden layers, but as the output layer is given certain specifications by the training

data the output layer usually has a different activation function. For now it suffices to know that activation functions are an important part of neural networks and we will more into detail with examples in section 3.3.

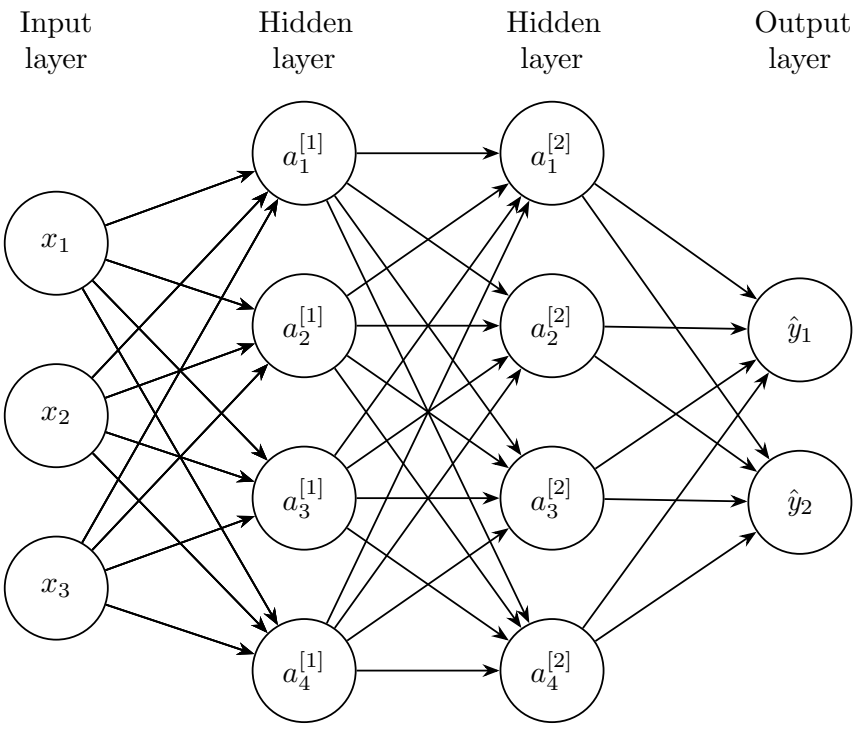


Figure 3.1: A fully connected neural network (FCNN) with 2 hidden layers with an input size of 3 and outputting two predictions $\hat{y}$. In a fully connected neural network each neuron is connected to all neurons in the next layer, while in a simpler FFNN a neuron might only be connected to a few of the neurons in the next layer. Note that the number displayed in square brackets denotes which hidden layer the neuron is located in, while the subscript denotes the neurons location in the layer.

Now that we have introduced the essentials of a neural network let's dive into the mathematics. A neural network consists of many neurons in several layers as described in fig. 3.1, but the mathematical formulas are the same for every neuron, given that they use the same activation function. Activation functions might differ between layers, but is usually the same for neurons in the same layer. In general the calculations in a neuron consists of a linear operation using weights to determine the relative importance of each input, and a non-linear transformation $\sigma_i(z)$. This non-linear transformation is what lets the neural network to represent a more complex problem than what linear and logistic regression are able to. The equations describing a forward pass in the neural network of fig. 3.1 can then be given as:

- $a_k^{[l]}$ is the activation of node k in layer l.

$$a_k^{[l]} = \sigma\left(\sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]}\right) \tag{3.1}$$

- $w_{jk}^{[l]}$ is the **weight** from node j in layer $l-1$ to node k in layer l.
- $b_k^{[l]}$ is the **bias** of node k in layer l.
- σ is a non-linear activation function.
- The weights w and the biases b are the parameters being trained during optimization.
- Note that we start counting the number of layers of a network from the first hidden layer, and that the first and last layer are given as:
  - $a_k^{[0]} = x_k$
  - $a_k^{[L]} = \hat{y}_k$
  - $n^{[l]}$ is the number of nodes in layer l.
  - The input dimension is $n_x = n^{[0]}$ and the output dimension is given as $n_y = n^{[L]}$.

The feedforward operation, also known as feedforward propagation, from the input layer to the first hidden layer is given by eqs. (3.3) and (3.4).

$$z_k^{[1]} = \sum_{j=1}^{n_x} w_{jk}^{[1]} x_j + b_k^{[1]} \tag{3.2}$$

$$= \sum_{j=1}^{n^{[0]}} w_{jk}^{[1]} a_j^{[0]} + b_k^{[1]} \tag{3.3}$$

$$a_k^{[1]} = \sigma\left(z_k^{[1]}\right) \tag{3.4}$$

$$\text{for } k = 1, \ldots, n^{[1]}. \tag{3.5}$$

Moving from one hidden layer to the next is equal is general for all hidden layers, given by:

$$z_k^{[l]} = \sum_{j=1}^{n^{[l-1]}} w_{jk}^{[l]} a_j^{[l-1]} + b_k^{[l]} \tag{3.6}$$

$$a_k^{[l]} = \sigma\left(z_k^{[l]}\right) \tag{3.7}$$

$$\text{for } k = 1, \ldots, n^{[l]}, \tag{3.8}$$

$$l = 1, \ldots, L-1. \tag{3.9}$$

Equations (3.6) and (3.7) can be written in vectorized form avoiding the use of indices and will then be given as

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]} \tag{3.10}$$

$$a^{[l]} = \sigma\left(z^{[l]}\right). \tag{3.11}$$

Going from the last hidden layer $L-1$ to the output layer, L, we have:

$$z_k^{[L]} = \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]} \tag{3.12}$$

$$a_k^{[L]} = o\left(z_k^{[L]}\right) = \hat{y}_k \tag{3.13}$$

$$\text{for } k = 1, \ldots, n_y, \tag{3.14}$$

$$= 1, \ldots, n^{[L]}. \tag{3.15}$$

Note that when moving to the output layer we call the activation function $o(\cdot)$ instead of the $\sigma(\cdot)$ used in the previous layers to signify that the activation function used in hidden layers is usually different from the output activation function.

After performing a forward pass, i.e. feedforward propagation, we want to optimize the neural network. As mentioned the parameters in a neural network that "learn" are the weights, $w_{jk}^{[l]}$, and the biases, $b_k^{[l]}$. In the linear regression example in 2.3 we saw that the optimization problem could be solved by a matrix inversion operation. This is no longer a possible solution method because of the non-linearity introduced by activation functions, which causes the weights and biases to not have closed-form first derivatives. Thus, we turn to the iterative method of gradient descent introduced in section 2.7. The cost function, C, will depend on the problem at hand, e.g. if the output is a real number we could use MSE as for linear regression or if the output is a probability a form of cross-entropy cost function is a possibility. The choice of

cost function is important to the problem we are solving, but does not change the gradient descent update of the weight and bias parameters given as:

$$w_{jk}^{[l]} \leftarrow w_{jk}^{[l]} - \eta \frac{\partial C}{\partial w_{jk}^{[l]}}, \tag{3.16}$$

$$b_{k}^{[l]} \leftarrow b_{k}^{[l]} - \eta \frac{\partial C}{\partial b_{k}^{[l]}}, \tag{3.17}$$

$$\text{for all } \begin{cases} j = 1, \ldots, n^{[l-1]} \\ k = 1, \ldots, n^{[l]} \\ l = 1, \ldots, L \end{cases} \tag{3.18}$$

where η is again the learning rate introduced in section 2.7, and the updated parameters are calculated for every neuron and neuron connection in the neural network (note gradient descent does not touch the input layer, i.e. the training data). This parameter update is done by what is known as the **backpropagation algorithm**.

3.2 Backpropagation

The backpropagation algorithm, sometimes abbreviated **backprop**, was discovered multiple times independently in the 1970s and 80s. The backprop algorithm is not only applicable to multi-layered neural networks, but can in general compute the derivative of any function, also functions where the correct derivative is to answer "undefined".

A brute force gradient descent would require us to calculate the gradient of each parameter in the entire neural network once per gradient descent update. The backpropagation algorithm uses the layered structure of the neural network in such a way that it can more efficiently compute gradients [23]. Backpropagation is sometimes mistaken as the actual learning algorithm of a neural network, but is in fact only the method for which we calculate the gradients, while the learning is performed by an algorithm like stochastic gradient descent (SGD) [11].

Computing the derivatives $\frac{\partial C}{\partial w_{jk}^{[l]}}$ and $\frac{\partial C}{\partial b_{k}^{[l]}}$ is done by recursive use of the chain rule. According to the chain rule the derivative of a function f dependent on a function g which is again dependent on x is given by $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}$. For a function f dependent on multiple functions g_i which are all dependent on x we add a summation and find the derivative as:

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}. \tag{3.19}$$

To calculate the derivatives in eqs. (3.16) and (3.17) we need to define the cost function C. We will use the previously mentioned mean squared error (MSE) function as our cost function. In the notation of our neural network the MSE function has the form

$$C = \frac{1}{2n} \sum_{x} \| y(x) - a^{[L]}(x) \|^2, \tag{3.20}$$

where n is the total number of training examples, the sum is over individual training examples, $y = y(x)$ is the corresponding desired output (the true output), L denotes the number of layers in the network, and $a^{[L]}(x)$ is the vector of activations outputted from the network when given the input x [28].

In order to use backpropagation we need two assumptions about the cost function to be satisfied. The first assumption is that the cost function can be written as an average $C = \frac{1}{n} \sum_x C_x$ over cost functions for each individual training example, x. This holds for the mean squared error, also known as the quadratic cost function, where the cost for a single example can be written as $C_x = \frac{1}{2} \| y - a^{[L]} \|^2$ [28]. The second assumption that must be satisfied is that the cost function can be written as a function of the output of the neural network. For the MSE cost function we can write

$$C = \frac{1}{2} \| y - a^{[L]} \|^2 = \frac{1}{2} \sum_{k} \left(y - a_k^L \right)^2, \tag{3.21}$$

where a_k^L are the output activations of the neural network and the assumption is satisfied. The cost function also depends on the desired true output y, but because each input x is fixed the corresponding true output y is also a fixed parameter. The true output y can in no way be changed or learned by modifying weights or biases in the network. As such it makes sense to only regard the cost function as a function of the output activations alone, where y is just used as a parameter to help define the function [28].

The backpropagation algorithm is mainly based on well-known linear algebra operations like vector addition, vector-matrix-multiplication, and so on. However, there is a not so well-known operation that is simplifies the notation of the backpropagation algorithm, namely the Hadamard product. Given two vectors or matrices of the same dimension the Hadamard product denotes the elementwise multiplication of the two vectors/matrices. Two vectors $\mathbf{s}$ and $\mathbf{t}$ we can write

$$\mathbf{s} \circ \mathbf{t} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \circ \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 4 * 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}. \tag{3.22}$$

Backpropagation is implemented for simple use in most ML applications which removes the need for a deep understanding of the individual computations performed by the backprop algorithm. However, having some sort of understanding of the overall goal of the algorithm and a higher-level knowledge of the main computations of the algorithm can help understand how neural networks operate and how they are optimized. The backpropagation algorithm is defined by four fundamental equations which will be presented and given a brief explanation. For more details on the derivations and proofs there exists a multitude of lecture notes going into more depth, among them chapter 2 in the book Nielsen [28].

Before we dive into the four fundamental equations we will introduce the quantity, $\delta_k^{[l]}$, which we call the *error* in the kth neuron in the lth layer. Backpropagation will give us a formula for how to compute the error and how it is related to the parameter updates described previously. If a small change $\Delta z_k^{[l]}$ is added to a neurons weighted input, $z_k^{[l]}$, so that the neuron outputs $\sigma(z_k^{[l]} + \Delta z_k^{[l]})$ instead of $\sigma(z_k^{[l]})$, that change will propagate through the network until it reaches the last layer where it will affect the final cost with a size equal to $\frac{\partial C}{\partial z_k^{[l]}} \Delta z_k^{[l]}$. Now we want to find a $\Delta z_k^{[l]}$ which will improve the cost, i.e. making it smaller. If the derivative, $\frac{\partial C}{\partial z_k^{[l]}}$, has a large magnitude, either positive or negative, then choosing a $\Delta z_k^{[l]}$ of opposite sign can reduce the cost by a significant amount. However, if $\frac{\partial C}{\partial z_k^{[l]}}$ is close to zero, then it will be very difficult to find a value for $\Delta z_k^{[l]}$ that improves the cost, and the neuron will in that sense be determined as close to optimal. If we were to change the activation of a neuron, $a_k^{[l]}$, instead of the weighted input, $z_k^{[l]}$, we would end up with a similar result, but with a more complicated presentation of backpropagation [28].

The first equation of backprop is an equation for calculating the error in the output layer.

$$\delta_k^{[L]} = \frac{\partial C}{\partial a_k^{[L]}} \sigma'(z_k^{[L]}), \tag{3.23}$$

where the term $\frac{\partial C}{\partial a_k^{[L]}}$ measures how quickly the cost function changes if the output activation of the kth neuron is changed. If the cost function does not depend much on the given neuron the error $\delta_k^{[L]}$ will be small. The second term, $\sigma'(z_k^{[L]})$, measures how quickly the activation function is changing at $z_k^{[L]}$. We can write the same equation in vectorized form as follows:

$$\delta^{[L]} = \nabla_a C \circ \sigma'(z^{[L]}), \tag{3.24}$$

where $\nabla_a C$ is a vector consisting of the partial derivatives $\partial C/\partial a_k^{[L]}$. If we use the MSE cost function previously introduced then $\nabla_a C = (a^{[L]} - y)$, and the final vectorized form of eq. (3.23) applied to the MSE cost function can be written as:

$$\delta^{[L]} = \left(a^{[L]} - y\right) \circ \sigma'(z^{[L]}). \tag{3.25}$$

The second fundamental equation of backprop computes the error $\delta^{[l]}$ with regards to the error in the next layer, $\delta^{[l+1]}$.

$$\delta^{[l]} = \left(\left(w^{[l+1]}\right)^T \delta^{[l+1]}\right) \circ \sigma'(z^{[l]}), \tag{3.26}$$

where $(w^{[l+1]})^T$ is the transpose of the weight matrix $w^{[l+1]}$ for the $l+1$th layer. If we know the error in the $l+1$th layer, then we can think of applying the transpose weight matrix as moving the error backwards in the network, giving us a measure of the error in the lth layer. By taking the Hadamard product $\circ\sigma'(z^{[l]})$ we move the error back through the activation function in layer l, and thus we have the error of the weighted input to layer l, $z^{[l]}$. By combining eqs. (3.23) and (3.26) we can compute the error $\delta^{[l]}$ for any layer in the network. First, using eq. (3.23) to compute $\delta^{[L]}$, then use eq. (3.26) to compute $\delta^{[L-1]}$, then eq. (3.26) again to compute $\delta^{[L-2]}$, and so on until the start of the network.

The third equation calculates the change in the cost function with respect to any bias in the network.

$$\frac{\partial C}{\partial b_k^{[l]}} = \delta_k^{[l]}, \tag{3.27}$$

thus meaning that the error $\delta_k^{[l]}$ is exactly equal to the rate of change in the cost function, $\partial C/\partial b_k^{[l]}$, which we already know how to calculate thanks to eqs. (3.23) and (3.26).

The final equation for backpropagation is used to compute the rate of change in the cost function with respect to any weight in the network. The equation is given by

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = a_j^{[l-1]} \delta_k^{[l]}. \tag{3.28}$$

This tells us how to compute the partial derivatives with respect to $\delta^{[l]}$ and $a^{[l-1]}$ which we know how to calculate. Equation (3.28) can be written in a possibly more intuitive manner, with no indices, as:

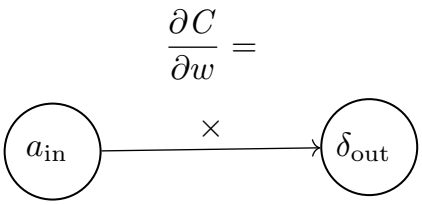


Figure 3.2: Zooming in on the two neurons connected by the weight w. Illustration is reproduced from Nielsen [28] chapter 2.

$$\frac{\partial C}{\partial w} = a_{\text{in}} \delta_{\text{out}}, \tag{3.29}$$

where a_{in} is the activation of the neuron input to the weight w, and δ_{out} is the error of the neuron output from the weight w. An illustration of the situation between the nodes connected by the weight w is presented in fig. 3.2.

From eq. (3.29) it follows that if the activation a_{in} is close to zero, $a_{\text{in}} \approx 0$, then the gradient term $\partial C/\partial w$ will also be small. When the gradient term is close to zero we say that the weight learns slowly, because during gradient descent the weight changes in very small increments. This means that weights from neurons with small activations will learn more slowly than weights for neurons where the activation is not small.

The term $\sigma'(z_k^{[L]})$ in eq. (3.23) can tell us something about how the weights in the output layer are able to learn. If the activation function of the output layer is very flat when $\sigma(z_k^{[L]})$ is close to 0 or 1, then the derivative $\sigma'(z_k^{[L]}) \approx 0$. The sigmoid activation function is just such a function, and will be described in more detail in section 3.3. The weights in the output layer will therefore learn very slowly if the activations of the output neurons are either very low ≈ 0 or very large ≈ 1. If this happens we say that the output neuron is *saturated*, and as a result the weight learns very slowly, if anything at all. The same applies for the corresponding bias of the output layer.

Similarly for any of layers previous to the output layer the term $\sigma'(z_k^{[L]})$ in eq. (3.26) can cause the error $\delta_k^{[l]}$ can become very small if the neuron is close to saturated. This will again cause the weights and biases input to the saturated neuron will learn slowly.

In conclusion, remembering and fully understanding all the equations like eqs. (3.23) and (3.26) to (3.28) is not very important, and not really to be expected without significant study, but they can teach us something about the learning taking place in a neural network. Namely, that a weight will learn slowly if the input neuron is low-activation, or if the output neuron is saturated, i.e. is high- or low-activation. Note also that the four fundamental equations here presented are general, and can be applied to any activation

function. This lets researchers create their own activation functions with specific properties that can help improve learning.

We list the four main equations of the backpropagation algorithm for an easy overview.

$$\delta_k^{[L]} = \frac{\partial C}{\partial a_k^{[L]}} \sigma'(z_k^{[L]}) \tag{Backprop 1}$$

$$\delta^{[l]} = \left(\left(w^{[l+1]} \right)^T \delta^{[l+1]} \right) \circ \sigma'(z^{[l]}) \tag{Backprop 2}$$

$$\frac{\partial C}{\partial b_k^{[l]}} = \delta_k^{[l]} \tag{Backprop 3}$$

$$\frac{\partial C}{\partial w_{jk}^{[l]}} = a_j^{[l-1]} \delta_k^{[l]} \tag{Backprop 4}$$

A high-level algorithmic implementation of the backpropagation algorithm after having completed a forward pass through the network can be written as:

1. **Output error:** Compute the vector $\delta^{[L]} = \nabla_a C \circ \sigma'(z^{[L]})$.
2. **Backpropagate the error:** For each layer $l = L-1, L-2, \ldots, 2$, compute $\delta^{[l]} = ((w^{[l+1]})^T \delta^{[l+1]}) \circ \sigma'(z^{[l]})$.
3. **Output:** Calculate the gradient of the cost function by computing $\frac{\partial C}{\partial b_k^{[l]}} = \delta_k^{[l]}$ and $\frac{\partial C}{\partial w_{jk}^{[l]}} = a_j^{[l-1]} \delta_k^{[l]}$.

3.3 Activation functions

Activation functions are an integral part of neural networks, and can have a significant impact to the effectiveness of learning. The activation functions introduces the all-important non-linearity needed for the neural network to be able to approximate any non-linear function that we want to learn. The different activation functions can have very different non-linearities, e.g. some activation functions are discontinuous while others have a zero-derivative for every point except where the input is zero. In this section we will present a selection of activation functions that have been used both historically and more recently in modern day deep learning.

3.3.1 Sigmoid and Tanh

The sigmoid and the hyperbolic tangent functions were used historically as activation functions in deep learning and neural networks, but have in recent

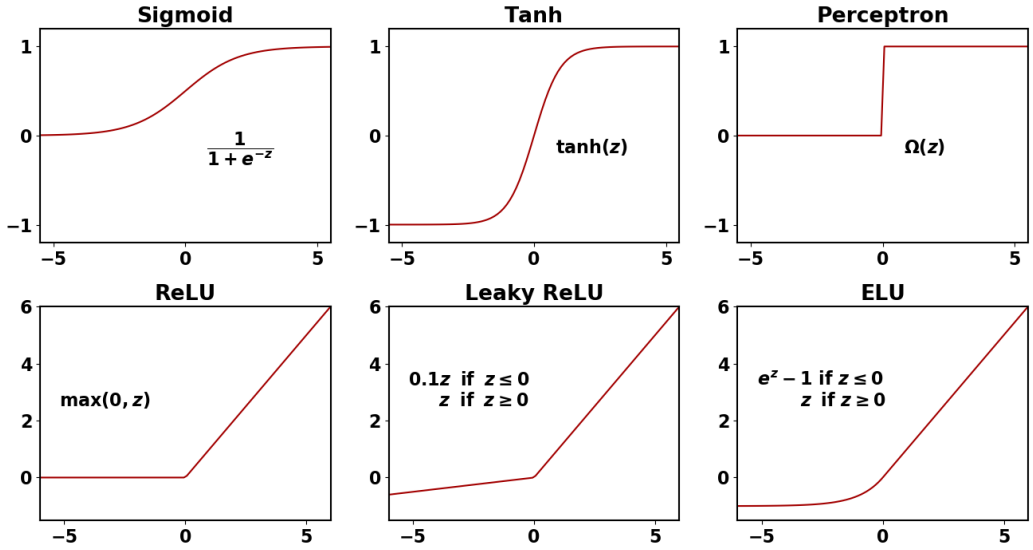


Figure 3.3: A range of popular activation functions. The top row have been used a lot historically, while the bottom row consists of modern specialized activations functions for neural networks. The perceptron function is a step-function where the derivative is discontinuous and not possible to train using gradient descent. Thus we will focus our attention on the other functions which will be presented in the following sections. Figure reproduced from Mehta et al. [23].

years seen a lot less use due to the improvements observed by new and better activation functions. Both functions are now mostly used for educational purposes, in addition to a few specialized models like recurrent neural networks (RNNs), many probabilistic models, and some autoencoders which cannot make use of piecewise linear activation functions [11].

The sigmoid function is mathematically defined as

$$\sigma = \frac{1}{1+e^{-z}}, \tag{3.30}$$

with a derivative that is cheap to compute given by

$$\frac{d\sigma}{dz} = \sigma(z)(1-\sigma(z)). \tag{3.31}$$

The hyperbolic tangent activation function is defined as

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}. \tag{3.32}$$

The derivative of the hyperbolic tangent is not as easily computed as the derivative of the sigmoid, given by

$$\frac{\mathrm{dtanh}(z)}{\mathrm{d}z} = 1 - \tanh^2(z). \tag{3.33}$$

The hyperbolic tangent and sigmoid activation functions introduce the required non-linearity to a NN, but the functions have a common drawback. Both functions are prone to saturated activation functions, thus leading to the derivative of the output to approach zero, $\partial\sigma/\partial z \approx 0$ for $z >> 1$. We can visually confirm this by looking at the plots of the sigmoid and hyperbolic tangent in fig. 3.3 where both functions are very flat for $z >> 1$, and as we know a function with a flat curve will have a derivative ≈ 0. This is what we described in section 3.2 when talking about saturated activation functions. Because both functions are bounded by $[0, 1]$ and $[-1, 1]$ we can avoid issues with "exploding gradients" using the sigmoid or hyperbolic tangent as activation function, but we run the risk of instead meeting saturated activation functions leading to *vanishing gradients* instead.

3.3.2 Rectified linear

Rectified linear units (ReLUs) were introduced in 2010 by Nair and Hinton [27] and have become the industry standard for NNs. ReLUs were popularized through their use in the famous AlexNet which revolutionized the ImageNet challenge in 2012. The ReLU function is defined such that it is zero for all negative inputs and exactly the same as the input for all positive values. Mathematically we can define the function as

$$\mathrm{ReLU}(x) = f(z) = \begin{cases} z, & \text{if } z > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3.34}$$

The corresponding derivative is then a very simple step-function given by

$$f(z) = \begin{cases} 1, & \text{if } z > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3.35}$$

ReLU as all the other activation functions we have discussed is monotonic, meaning that the function always moves in the same direction, or stays at the same value. I.e. the function does not go from increasing to decreasing in value as we move along an axis, but can go from increasing value to staying the same and then continuing to increase.

Note that ReLU is not differentiable at $z = 0$, which might seem to invalidate the use of ReLU as an activation function together with learning algorithms based on gradient descent. However, in practice gradient based methods perform well enough for ReLU functions to be used in ML, partly

due to the fact that NNs usually don't reach a local minima of the cost function, but instead just reduces the cost functions value significantly [11].

A variety of ReLU activation functions have been suggested, among them the leaky rectified linear unit (LReLU) which introduces a small slope to the negative part of the activation, and is defined as

$$\text{LReLU}(x) = f(z) = \begin{cases} z, & \text{if } z > 0 \\ \alpha z, & \text{otherwise,} \end{cases} \tag{3.36}$$

where α is a chosen parameter usually given a small value like 0.01 or 0.1. The derivative is then given as

$$f(z) = \begin{cases} 1, & \text{if } z > 0 \\ \alpha, & \text{otherwise.} \end{cases} \tag{3.37}$$

ReLU activation functions avoids the issue of saturated activations causing vanishing gradients. For activation functions like ReLU, LReLU, and ELU (presented in fig. 3.3) the gradients stay finite even for large inputs, thus avoiding exploding gradients.

In addition to a huge variety of possible activation functions there exists methods of regularization like gradient clipping which can be employed to avoid issues of exploding gradients, but this can come with side-effects like slowing down training.

3.4 Universal approximation theorem

A linear model that maps input features to outputs via matrix multiplication can by definition only represent linear functions. Training such a model would be relatively easy, but unfortunately we often want to learn non-linear functions when working with neural networks [11].

Initially it might seem like we have to develop a new specific model for every kind of non-linear function approximation we want to learn. Luckily this is not the case and the **universal approximation theorem** first presented by Hornik et al. [13] states that a standard feedforward neural network given sufficiently many hidden units is capable of approximating any measurable function to any non-zero level of error [13]. The first iteration of the universal approximation theorem were initially proven with activation functions that were very prone to saturation, but has since been proven for a wide range of activation function classes, including the popular rectifier functions [20].

A consequence of the universal approximation theorem is that any failure in whatever application we utilize a neural network must be a result of inadequate learning, too few hidden units (neurons), or that the deterministic

relationship between input and output is not strong enough [13]. Meaning, if the features we use as input are insufficiently correlated with a specific output the model will not robustly give the correct output compared to the true target. Thus, we know that whatever function we want to approximate in order to solve a given task can be *represented* by a large NN, but we are not guaranteed that the learning algorithm will be able to *learn* the function. The universal approximation theorem does not state how large a specific NN must be in order to approximate a given function, only that there exists one. It is therefore up to us to choose the number of neurons and layers to use in our function approximation. Too large of a network can easily lead to overfitting a function and thus generalize poorly, while a very small network might not be able to represent the target function at all.

3.5 Network architecture

In section 3.4 we learned that any function can be approximated with a neural network. However, choosing what that NN should look like is not as easy, and will in part be dependent on the task at hand.

A neural network of one single hidden layer can in theory approximate any function, but for a very complex function we might need an infeasible number of hidden neurons in that layer, making learning and generalization very difficult. Instead of using a single very large layer we can construct a neural network consisting of multiple hidden layers with fewer hidden neurons in each layer. In many cases this can reduce the amount of neurons needed significantly, in addition to reducing the generalization error of the trained network [11]. When increasing the number of hidden layers we often say that we are increasing the *depth* of the network, and it has been found empirically that for a wide variety of tasks increasing the depth of neural networks results in better generalization [11].

Besides choosing the number of neurons in each layer along with the depth of a neural network there exists other architectural considerations that are worth mentioning. Entire neural network architectures have been developed with a specific task in mind, e.g. convolutional neural networks (CNNs) which are widely used in image recognition and computer vision tasks. A feedforward neural network can be generalized to a recurrent neural network (RNN) which is very commonly used when working with sequential data like speech recognition and sentiment analysis. CNNs and RNNs have seen tremendous success in a great variety of applications, but as we have not applied either type of neural network in this project we will not give further explanations. Instead we want to point the interested reader to chapter 9 in Goodfellow et al. [11] as well as chapter 10 in Mehta et al. [23] for details on CNNs, and chapter 10 of Goodfellow et al. [11] for a thorough introduction

to RNNs.

So far we have worked with neural networks were every neuron in one layer is connected to every neuron in the next layer, but for more specialized networks this is not always the case. In certain situations it can be beneficial to reduce the number of connections in a network, e.g. in order to reduce the complexity of the model and avoid overfitting or to reduce the amount of parameters and thus computations that are required. This can be done previous to training, which would then result in a standard FFNN, not a FCNN, or it can be done during training which is a deep learning regularization technique known as *dropout*, where neurons in the neural network are dropped randomly during training.

3.6 Pre-trained networks

Some tasks might be too complex to directly train a model, e.g. the model is too complex, hard to optimize, or the task can be very difficult. If we see that learning initially fails we can train the model on a less complex problem then move on to the more complex problem and continue training. Another form of pre-training is to first train a simple model to solve a problem, and then increase the complexity of the model. Generally pre-training entails that we train simple models on simple problems before training the desired model on the desired task we want to solve [11].

A related method to pre-training is transfer learning, where a model that is trained on a different task can be used as the starting model before training the model to the actual task to be solved. E.g. a model trained on the ImageNet classification task can be used as a starting point of other image classification tasks on different datasets [11].

Chapter 4

Reinforcement Learning

Learning by interacting with the environment around us is probably the most basic and common way of learning we observe around us. Infants waving their arms around have no teacher that can tell them what will happen, but they are able to observe their environment and what happens when waving. This method of learning cause and effect, by observing what consequences certain actions result in is a method of learning not only specific to infants, but also something that can be observed in adult behavior. Learning to drive, how to act in social settings, or how to play football are all instances of situations where theoretical knowledge might help, but is in no way enough to master the given skill.

Reinforcement learning (RL) is a *computational* approach to learning by interacting with an environment and observing the outcome of actions. The goal is to learn how to map situations to actions to maximize a numerical reward, without specifying which actions to take. Instead the learner must discover what actions results in the greatest reward by trial-and-error. In more complex tasks an action might not just affect the immediate reward, but also have significant impact on the subsequent rewards in the next situations. In reinforcement learning the learner is most often called an *agent*.

A learning agent must to some extent be able to observe the state of its environment and must be able to apply actions that affects the state of the environment. The agent must also have a goal that is related to the state of the environment, e.g. winning a chess game or making a car complete one lap of a racing game. A Markov decision process is supposed to only include the three components, sensation, action, and goal, in their simplest form without trivialization [38].

If the reader is interested in further explanations and details than what is provided in this chapter we recommend the freely available PDF-version of the book *Reinforcement Learning* by Sutton and Barto [38].

In this chapter we will give a high-level introduction to reinforcement

learning, then we will describe some of the differences between reinforcement learning and other machine learning methods, before we present the RL learning method implemented, namely the proximal policy optimization method.

4.1 Introduction

In *supervised learning* learning happens through training sets of labeled where each label corresponds to a correct action, e.g. returning the correct category to which the input data belongs to. In such learning the goal is to train a model that can generalize to unseen examples and is still able to find the correct "actions" that correctly label those examples. In an interactive environment obtaining examples of every possible situation in order to find the correct actions for every representative state of the environment would be impractical, if not impossible. Instead the agent must learn from its own experiences [38].

Reinforcement learning also differs from *unsupervised learning* where the goal is generally to find structures in unlabeled data, while for RL we try to maximize a reward rather than finding structures in data. Although it sounds natural that *supervised* and *unsupervised learning* would be exhaustive in describing machine learning paradigms, reinforcement learning is considered to be a third machine learning paradigm.

All reinforcement learning examples will include some *interaction* between an active decision-making agent and an environment. The agent tries to achieve a *goal* despite there being *uncertainty* in how the agent sees the environment. The agents actions are allowed to affect the future states of the environment, thus affecting what actions will be available in the future. These actions can usually not be fully predicted, meaning that the agent must frequently observe the environment to take the best possible actions [38].

A challenge we face that is specific to RL is the trade-off between *exploration* and *exploitation*. In order to maximize the reward an agent must base its actions on actions already known to give high reward, but the agent must first find those actions. The agent must exploit good actions it has found, but at the same time keep exploring in order to find better actions in the future. Balancing this trade-off is not easy, and choosing to focus only on exploration, or only on exploitation will create an agent unfit to solve the task at hand. The agent should try a variety of actions and as training progresses favor the actions that appear to give the best reward.

4.1.1 Reinforcement learning systems

Besides an agent and an environment, there are a few core elements that are present in a reinforcement learning system: a *policy*, a *reward signal*, a *value function*, and optionally a *model* of the environment [38].

A *policy* defines how the learning agent acts at a given time. In some ways the policy works as a mapping between the state of the environment as perceived by the agent to the best actions for those states. The policy can be everything from a simple function or lookup table to a much more complex search process. The policy is thus the core of the learning agent, as the policy is enough to determine the agents behavior [38].

The *reward signal* defines the goal of the RL task at hand. For each update step of the agent the environment will send a single value to the agent, the *reward*. The goal of the agent is to maximize the total reward received, not just the instantaneous reward, but also maximizing future rewards. The reward signal defines what is good and what is bad actions, based on how each action has affected the environment, either in a positive or negative way. If an action receives a low reward the policy of the agent may be changed in such a way that it selects different actions the next time it encounters similar states. The reward is thus the primary source of changes in policy, and the definition of the reward signal can significantly impact the effectiveness of learning [38]. A human example of reward signals are pleasure and pain. If we do something good, like eating a good piece of food we receive pleasure, while if we cut our hand we instantly feel pain.

A *value function* indicates what is good in the long run. While the *reward signal* gives an immediate feedback, the *value function* gives an approximation of the expected accumulated reward received from the current state of the environment and onwards. The reward will thus give an estimation of the desirability of an instant state of the environment, while the *value function* indicates the long-term desirability of states taking into account the states that are likely to follow and the rewards of those states. Thus, a state can at the same time have low reward and high value, if it leads to several desirable states, although the state itself is not as instantly desirable. The opposite case of a singular desirable state of high reward, but low value is of course also possible [38]. A human example of value could be that of a student taking a course at university that he isn't very fond of, but although his current circumstances have low instant satisfaction (low reward), he recognizes that he must complete the course and his degree before achieving his dream job (high reward over time).

Note that there could be no *value function* without *reward signals*, but even so we are most concerned with the values when we are making and evaluation actions. We want the agent to take the actions that leads to the states of highest value, not necessarily highest reward, because those actions will

result in the highest total accumulated reward. It is however much harder to estimate values than it is to determine an instant reward of a state. The reward is often given directly from the environment, while the value must be estimated and re-estimated continuously during training [38].

The final core element found in some RL systems is a *model* of the environment. A model mimics the behavior of the environment, giving the agent the possibility to predict the next state, given a state and action. Models are used to *plan* actions in advance, by considering multiple possible future states and actions before they are actually experienced by the agent. Reinforcement learning methods utilizing models and planning are called *model-based* methods, while *model-free* methods rely fully on trial-and-error learning [38].

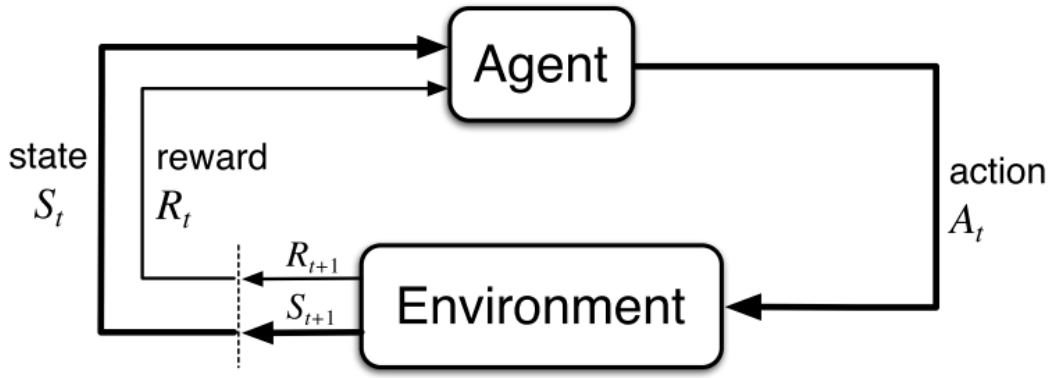


Figure 4.1: A general visualization of the interaction between a reinforcement learning (RL) agent and its environment. Reproduced from figure 3.1 of Reinforcement Learning by Sutton and Barto [38]. Licensed under CC BY-NC-ND 2.0.

4.2 Computing the policy gradient

Model-free methods, as described in subsection 4.1.1, are further distinguished between *value-based* and *policy-based* methods [38]. Both methods aim to maximize the expected reward which for policy-based methods is done by directly optimizing the decision policy, while value-based methods learns to estimate the expected value of a state-action pair optimally and in turn determine the optimal action for each state [8].

In this section we will look at the policy gradient and how we can compute it, before we in section 4.3 go into more specific derivations and explanations of the proximal policy optimization (PPO) method which is the RL method utilized in the simulations of the project.

Following the derivation of the policy gradient method given in appendix C of Rabault et al. [33] we have a policy function, π_Θ, represented by an artificial neural network (ANN) where all weights are collectively given by the variable, Θ. We can formulate the learning problem as finding the optimal weights of the ANN such that the expected accumulated reward is maximized. In mathematical notation expressed as:

$$R_{max} = \max_\Theta \mathbb{E}\left[\sum_{t=0}^{H} R(s_t)|\pi_\Theta\right], \tag{4.1}$$

where π_Θ is the policy function described by the ANN with weights, Θ, and where s_t is the state of the system at time, t. The maximization of eq. (4.1) is solved through gradient descent methods optimizing the weights, Θ, of the ANN following experimental sampling of the system through interactions with the environment.

Introducing τ as a sequence of state-action-rewards triplets, $\tau = (s_0, a_0, r_0), (s_1, a_1, r_1), \ldots, (s_H, a_H, r_H)$ and overloading the R operator as $R(\tau) = \sum_i \gamma^i r_i$, then the value function obtained with the weights Θ can be written as

$$V(\Theta) = \mathbb{E}\left[\sum_{t=0}^{H} R(s_t, u_t)|\pi_\Theta\right] = \sum_\tau \mathbb{P}(\tau, \Theta)\, R(\tau). \tag{4.2}$$

The value function defined in eq. (4.2) is the quantity we want to maximize to find the optimal policy. To update the parameters we need to find the gradient of the value function with respect to the policy parameters, i.e. for DRL the weights of the neural network.

$$\nabla_\Theta V(\Theta) = \sum_\tau \nabla_\Theta \mathbb{P}(\tau, \Theta)\, R(\tau) \tag{4.3}$$

$$= \sum_\tau \frac{\mathbb{P}(\tau, \Theta)}{\mathbb{P}(\tau, \Theta)} \nabla_\Theta \mathbb{P}(\tau, \Theta)\, R(\tau) \tag{4.4}$$

$$= \sum_\tau \mathbb{P}(\tau, \Theta) \frac{\nabla_\Theta \mathbb{P}(\tau, \Theta)}{\mathbb{P}(\tau, \Theta)} R(\tau) \tag{4.5}$$

$$= \sum_\tau \mathbb{P}(\tau, \Theta)\, \nabla_\Theta \log\left(\mathbb{P}(\tau, \Theta)\right) R(\tau). \tag{4.6}$$

The last expression, eq. (4.6), represents a new expected value which can be sampled under the policy π_Θ and used as input to the gradient descent optimization. We need to calculate the log-prob gradient $\nabla_\Theta \log(\mathbb{P}(\tau, \Theta))$, which can be done as

$$\nabla_\Theta \log\left(\mathbb{P}(\tau,\Theta)\right) = \nabla_\Theta \log\left[\prod_t \mathbb{P}\left(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}\right)\, \pi_\Theta\left(a_t^{(i)}|s_t^{(i)}\right)\right] \tag{4.7}$$

$$= \nabla_\Theta \left[\log\, \mathbb{P}\left(s_{t+1}^{(i)}|s_t^{(i)}, a_t^{(i)}\right) + \sum_t \log\, \pi_\Theta\left(a_t^{(i)}|s_t^{(i)}\right)\right] \tag{4.8}$$

$$= \nabla_\Theta \sum_t \log\, \pi_\Theta\left(a_t^{(i)}|s_t^{(i)}\right), \tag{4.9}$$

where the last equation, eq. (4.9), is only dependent on the policy and not the dynamic model. This allows effective sampling and gradient descent to be carried out.

4.3 Proximal Policy Optimization - Background

Proximal policy optimization (PPO) presented by Schulman et al. [37] has shown itself to be very successful in a wide range of RL applications. The PPO algorithm combines the data efficiency and reliable performance of trusted region policy optimization (TRPO), while using only first-order optimization [37].

4.3.1 Policy Gradient background for PPO

A policy gradient method computes an estimator of the policy gradient and then use this value in a stochastic gradient ascent algorithm. A commonly used gradient estimator is given as

$$\hat{g} = \hat{\mathbb{E}}_t\left[\nabla_\theta \log \pi_\theta\left(a_t \,|\, s_t\right) \hat{A}_t\right] \tag{4.10}$$

where π_θ is a stochastic policy and $\hat{A}_t$ is an estimator of the advantage function at time t. The expectation value, $\hat{\mathbb{E}}_t$, given above is the empirical average over a finite batch of samples in an algorithm that alternates between sampling and optimization [37]. Implementations using automatic differentiation software obtains the gradient estimator $\hat{g}$ by constructing an objective function, L^{PG}, whose gradient is the gradient estimator. The objective function thus has the form

$$L^{PG}\left(\theta\right) = \hat{\mathbb{E}}_t\left[\log \pi_\theta\left(a_t \,|\, s_t\right) \hat{A}_t\right]. \tag{4.11}$$

It might be tempting to perform multiple steps of optimization on the loss L^{PG} using the same trajectory, however this often leads to destructively large policy updates [37].

4.3.2 Trusted Region Methods

In TRPO introduced by Schulman et al. [36] an objective function, also known as the "surrogate" objective is maximized subject to a constraint on the size of the policy update. Given by

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t \mid s_t\right)} \hat{A}_t \right] \tag{4.12}$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t \left[\mathrm{KL}\left[\pi_{\theta_{\text{old}}}\left(\cdot \mid s_t\right), \pi_\theta\left(\cdot \mid s_t\right) \right] \right] \leqslant \delta. \tag{4.13}$$

Where KL is shorthand for the Kullback-Leibler divergence, an asymmetric distance metric for comparing two probability distributions. θ_{old} is the vector of policy parameters before the update. The problem can then be solved approximately using the conjugate gradient algorithm after making a linear and a quadratic approximation on the objective and the constraint, respectively [36]. The theory behind TRPO originally suggests using a penalty instead of a fixed constraint, i.e.

$$\underset{\theta}{\text{maximize}} \quad \hat{\mathbb{E}}_t \left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t \mid s_t\right)} \hat{A}_t - \beta\, \mathrm{KL}\left[\pi_{\theta_{\text{old}}}\left(\cdot \mid s_t\right), \pi_\theta\left(\cdot \mid s_t\right) \right] \right] \tag{4.14}$$

However, choosing a value for the coefficient β that works well across multiple applications or within a single problem where the characteristics change during learning is very difficult. Hence, experiments have shown that using the penalized version of TRPO as a first-order algorithm emulating the monotonic improvement of TRPO is not a good enough solution.

4.4 Clipped Surrogate Objective - PPO

TRPO seeks to maximize the "surrogate" objective

$$L^{\text{CPI}}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta\left(a_t \mid s_t\right)}{\pi_{\theta_{\text{old}}}\left(a_t \mid s_t\right)} \hat{A}_t \right] = \hat{\mathbb{E}}_t \left[r_t(\theta)\, \hat{A}_t \right]. \tag{4.15}$$

We have here introduced the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$ such that $r_t(\theta_{\text{old}}) = 1$. $r(\theta) = 1$ means no change in the policy, while $r_t(\theta) \neq 1$ measures the difference between the old and the new policy. The superscript of the objective, CPI, refers to CPI introduced by Kakade and Langford [15]. If we were to maximize the objective, L^{CPI}, without any constraint or penalty the policy updates would become too large. In order to avoid such issues the objective is modified such that changes in the policy that would move $r(\theta)$

away from 1 are penalized [37]. The modified objective introduced for the PPO algorithm is given as

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta) \hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t \right) \right] \tag{4.16}$$

where ϵ is a hyperparameter of a given value, $\epsilon > 0.0$. Through experiments, a value of $\epsilon = 0.2$, proved to give the best results in the chosen test cases, and is also the value used in the code for the fluidic pinball project. The first term of the min function is equal to L^{CPI}. The second term of the min function, $\text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t$, clips the probability ratio such that $r_t \in [1-\epsilon, 1+\epsilon]$. The final objective is then a found as a lower bound on the unclipped objective by taking the min of the clipped and the unclipped objective [37].

Note that in the paper by Schulman et al. [37] an additional objective function using a penalty coefficient rather than applying clipping in the objective function is also presented. Through experiments it was found that the *clipped surrogate objective* version of PPO performed better than the penalty coefficient variation, but for those interested in how the Kullback-Leibler divergence is applied for PPO we refer to section 4 of Schulman et al. [37].

Chapter 5

Active Flow Control (AFC)

The problem of actively controlling fluid flows has been an area of research since the early 1900's when Prandtl [30] discovered the boundary layer. However, more passive methods of controlling a flow has been present for millennia, e.g. attaching feathers on arrows to stabilize the flight path [2]. In the years prior and during the second world war and the cold war a lot of research was put into flow control, mostly in military related flow systems [4].

Flow control strategies can be classified into three main categories: shape optimization, passive-, and active flow control [2]. In shape optimization the aerodynamic shape of an object is tailored to achieve a given goal, e.g. reducing the drag of a vehicle to increase fuel efficiency. Passive flow control involves a small change to the original configuration, like the addition of vortex generators to suppress flow separation by enhancing mixing between the boundary layer and the free flow [29]. The final method of flow control, which will be our focus, is active flow control (AFC) where an active control device is used to change the flow system to a more desirable state.

For active control we can differentiate between open- and closed-loop control. Open-loop active flow control will involve pre-determined controls, which are applied with no regard to the state of the flow. On the other hand, for closed-loop control the actuations of the controller are informed of the state of the flow by sensors or similar tools that can relate information of the flow to the controller [2]. For more details on open- and closed-loop control systems we refer to section 3 of Collis et al. [4].

The chapter will consist of a brief introduction to different methods of linear flow control. Gradient based control methods will also be presented, before we discuss the use of machine learning (ML)/artificial intelligence (AI) methods to control a flow system.

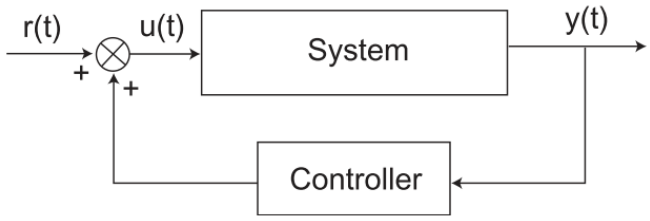


Figure 5.1: A typical closed-loop block diagram. The controller determines some actuation to be taken, dependent on the state of the system. Reprinted from Progress in Aerospace Sciences, 40, S. Scott Collis, Ronald D. Joslin, Avi Seifert, and Vassilis Theofilis, Issues in active flow control: Theory, control, simulation, and experiment, 237-289, Copyright (2004), with permission from Elsevier.

5.1 Linear control

5.1.1 Introduction and LQR

Many systems of interest where we want to apply control are either linear, or correspond to the linearization of a non-linear system. For a multitude of linear problems the *optimal* control is known. Linear control has several applications within fluid mechanic, among them stabilization of unstable laminar boundary layers. Although the original system might not be linear, controllers able to successfully stabilize the flow will change the system so that it increasingly approximates the linearization [7]. We will mention a few linear control methods, while referring to Duriez et al. [7] for details on the different methodology.

If the full state of the system can be measures a full-state feedback control method can be considered. For high-dimensional systems it might be unrealistic to measure the full state, but by using methods like a Kalman filter, an estimate of the full state can be found.

An LQR is a full-state feedback controller that minimizes a quadratic cost function to regulate the system of interest. Solving for the LQR controller is done by solving a Riccati equation which scales with the cube of the state dimension. Thus, for larger systems it will be too computationally intensive to solve for the controller simultaneously as the flow develops [7].

5.1.2 Sensor estimation and Kalman filtering

For many systems measuring the full state of the system is not technologically possible or is prohibitively expensive. In experiments where particle image velocimetry (PIV) is uses the full state of the system is measurable, but

such measurements are not practical for in-practice applications outside an experimental setting [7]. However, *estimating* the full state of the system is possible from limited and noisy sensor measurements. Such a method balances predictions created by a model with the sensor measurements to create a balanced estimate of the full state. Under well-defined conditions it is possible to create an estimator that converges to an estimate of the full state which can then be used with the optimal full-state feedback LQR control law.

The Kalman filter is possibly the most popular algorithm to estimate full-state of a system, and have been used in a great variety of applications [7]. The Kalman filter uses knowledge from noisy sensor measurements, actuation input, and a model of the process dynamics to estimate the full-state of the system. The Kalman filter is an *optimal* full-state estimator, and finding the optimal solution is often called linear quadratic estimation (LQE). As for the optimal controller of LQE, the optimal Kalman filter is found by solving another Riccati equation.

By combining the estimated state found by the Kalman filter with the optimal controller of LQR, i.e. combining the solutions of two separate Riccati equations, we obtain the optimal linear quadratic Gaussian (LQG) controller. The optimal LQG controller is thus not dependent on full-state measurements, but instead uses the full-state estimate of the Kalman filter [7].

5.1.3 Reduced order model (ROM)

As the Riccati equations scale with the cube of the dimension, $\mathcal{O}(n^3)$, high-dimensional linear systems will introduce unwanted latency in the control loop. This latency can prevent real-time control actuations of high-dimensional systems where the system develops quickly, which is very common in fluid flows. Reduced order models (ROMs) are used to avoid such issues by attempting to represent a complex system in a less complex manner, without loosing too much information. For flows of low *Re* or for weakly unstable flows this works quite well, but a minor difference in the initial condition or a small perturbation in the system can result in significantly different solutions. However, for certain systems ROMs in combination with LQG controllers have been successfully applied to control global oscillations in a separated boundary layer and to reduce the skin-friction drag in a channel flow [2].

5.2 Gradient-based and stochastic control

Although finding an optimal controller is the end-goal of a flow control problem, it is very often not a realistic goal for problems more complex than linear systems. Gradient-based control is a non-optimal control method based on calculating the gradient of an objective function with respect to the control. If

the gradient of the objective function is not possible to obtain, e.g. the objective function is not differentiable, we must use other methods like a stochastic method. This is a key advantage of stochastic control methods, but stochastic methods typically need many more evaluations of the objective function than gradient-based methods, in addition to scaling very poorly with the dimension of the parameter space. The method of minimizing the objective functions can vary, but one relatively common method implemented in both Muldoon [26] and Min and Choi [24] is solving a set of adjoint equations, which in these cases are derived from the Navier-Stokes equations.

In Muldoon [26] the objective function is found by solving the unsteady Navier-Stokes and energy equations over significant time periods, resulting in an objective function that is very costly to compute. The cost of computing the gradient of the objective function is equal to the cost of computing the objective function itself. A gradient-based method is thus more efficient because computing the objective function and the gradient a small amount of times are less costly than having to compute the objective function many times.

In gradient-based and stochastic control methods finding the best objective function might not be as straightforward as one may think. For example, in Min and Choi [24] a gradient-based control method was used to control vortex shedding behind a circular cylinder by blowing and suction control actuations. They applied three different objective functions, where the first objective function (called cost functional in the paper) is the pressure drag of the cylinder. The second objective function is the square of the difference between a target pressure and the real pressure on the cylinder surface, while the third objective function is the square of the pressure gradient at the surface of the cylinder. Each objective function is then minimized (objective 1 and 2) or maximized (objective 3) with respect to the blowing-suction actuations. Intuitively we might think that the objective function that explicitly includes the drag of the cylinder should be the best objective to reduce the drag on the cylinder, but Min and Choi [24] found that the second objective function with a given maximum of actuations was the most effective control strategy for reducing drag on the cylinder.

5.3 Deep reinforcement learning for AFC

The *fluidic pinball* system that we will simulate in chapter 10 and **??** is quite similar to the system of Min and Choi [24], most of all because both systems are governed by the Navier-Stokes equations. Both systems also experience vortex shedding for $Re = 100$, while the goal is to control the vortex shedding in order to reduce the drag of the system. As such, it would be possible to use similar methodology where objective functions based on the Navier-

Stokes equations are used to compute a gradient with respect to actuations. However, such a system would include not only computing the solutions of the Navier-Stokes equations, while also computing expensive objective functions and their gradient.

By using a DRL agent as our flow controller we still have to solve the Navier-Stokes equations to determine the evolution of the flow with respect to actuations, but instead of computing expensive objective functions and gradients, we use gradient descent to optimize the policy of our DRL agent. In Rabault and Kuhnle [32], it was found that 99.7% of the time spent in simulations was spent computing the fluid dynamics, and the time spent optimizing the agent is thus very small in comparison. However, a DRL agent is dependent on running simulations many times over, where it is allowed to learn by trial-and-error to find the best control actuations. This means that although the isolated optimization of the DRL agent is very effective in comparison to optimizing an objective function using classical gradient-based methodology, the total time spent on simulations is not necessarily in favor of DRL methodology.

Another reasoning behind the choice of using DRL agents to control the flow rather than more classical methods is that the goal of the project is not necessarily to find the best control strategy possible for the fluidic pinball system. The main goal is to investigate whether ML and AI models can be used at all in problems related to active flow control, and how increasing the complexity of the simulated system might affect the DRL agents ability to learn control strategies. Once it has been established that ML models actually are able to control flow systems, then it becomes more relevant to directly compare the effectiveness of the control with a much wider variety of control methods, such as gradient-based or ROM methods.

Chapter 6

Literature Review

The work of this thesis continues the work done by Rabault et al. [33] and Rabault and Kuhnle [32]. Therefore, we will give brief presentation of those papers as an introduction to the methodology applied, in addition to the theory that has been presented in previous chapters, and the implementation that will be explained further in part II. The first paper introduces the use of deep neural networks and reinforcement learning to the field of AFC, where a DRL agent is used to control actuations of two synthetic jets on a single cylinder.

The second article presents a framework for running AFC simulations controlled by DRL agents in parallel, applying the methodology to the same flow control problem as presented in the first article. With near perfect parallel scaling this methodology allows more computationally intensive problems to be researched, among them the fluidic pinball system which will be presented in chapter 10.

6.1 Artificial neural networks trained through deep reinforcement learning discover control strategies for active flow control

The simulation environment is a 2 dimensional system simulated at $Re = 100$. The system consists of a single cylinder of non-dimensional diameter $D = 1$ immersed in a box of non-dimensional length $L = 22$ and height $H = 4.1$. The cylinder is placed slightly off-center to encourage the creation of vortex shedding behind the cylinder. The unstructured computational mesh is created with the `gmsh` software of Geuzaine and Remacle [9], consisting of 9262 triangular elements, and the numerical timestep used is $dt = 5 \times 10^{-3}$.

A PPO agent is then trained to perform active flow control on the environment by controlling two synthetic jets placed on the top and bottom of

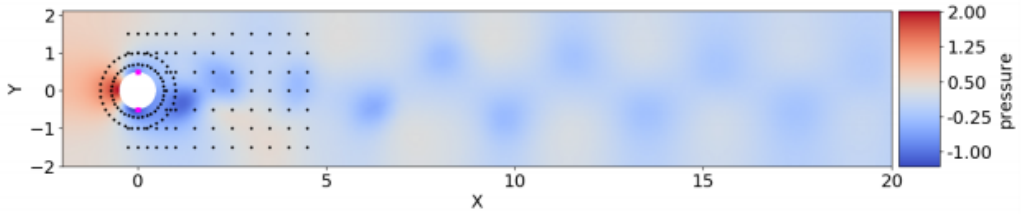


Figure 6.1: Pressure wake of the flow after initialization without active control. The black dots correspond to velocity probes, while the two red dots indicate the control jets. The figure corresponds to Figure 1 in [33]. The figure is reproduced from the freely available preprint available at https://arxiv.org/abs/1808.07664v5, licensed under CC BY 4.0 at ArXiv.

the cylinder, as shown in fig. 6.1. The placement of the jets on the north- and south pole of the cylinder ensures that the observed changes in the flow comes as a result of indirect flow control, and not a direct addition of momentum to the flow. The control is also configured such that the flow rate, $Q_i, i = 1, 2$, of the cylinders follows $Q_1 + Q_2 = 0$, i.e. the control will not add or subtract mass to the system.

The PPO agent is then trained by interacting with the environment during *episodes*, i.e. a limited amount of time, before the agent evaluates the effectiveness of the control in the previous episode, and then starts over with a new episode. The instantaneous reward function of the PPO agent is defined as

$$r_t = -\langle C_D \rangle_T - 0.2|\langle C_L \rangle_T|, \tag{6.1}$$

where $\langle \cdot \rangle_T$ indicates the sliding average back in time corresponding to one vortex shedding cycle. The agent tries to maximize the function, r_t, thus minimizing the drag coefficient and the mean lift. The PPO was not able to learn if it was allowed new actions for every numerical timestep. Instead, by only updating the action every 50 numerical timestep, and making the control continuous in time the agent was able to learn effective control. The continuous control at each numerical timestep can be defined as $c_{s+1} = c_s + \alpha(a - c_s)$, where c_{s+1} is the new control, a is the action given by the PPO agent for the current 50 timesteps, and $\alpha = 0.1$ is a numerical smoothing parameter.

Using these tricks the agent was able to learn a control strategy after approximately 200 episodes, corresponding to 1300 vortex shedding periods or 16000 sampled actions. For a finely tuned and stable control strategy the agent needed around 350 episodes. The mean drag coefficient of the baseline simulation without control is $\langle C_D \rangle \approx 3.205$, compared to $\langle C_D \rangle \approx 2.95$ of the flow with applied flow control. The PPO agents strategy is thus able to reduce the mean value of the drag coefficient by approximately 8%. The mean drag value of a hypothetical flow without any vortex shedding is $\langle C_D \rangle \approx 2.93$,

i.e. the flow control is able to suppress $\approx 93\%$ of the drag increase observed compared to a hypothetical flow that is kept completely stable. Representative snapshots comparing the baseline flow with the controlled flow are presented in fig. 6.2, where the effect of control can be observed clearly.

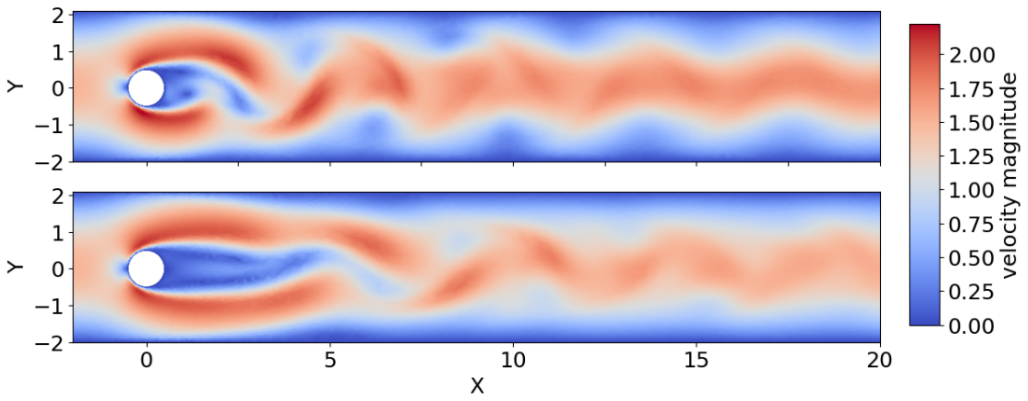


Figure 6.2: The top figure is a typical snapshot of the flow without any applied control, while the figure underneath is a representative snapshot of the flow with active flow control. The figure corresponds to Figure 4 in [33]. The figure is reproduced from the freely available preprint available at https://arxiv.org/abs/1808.07664v5, licensed under CC BY 4.0 at ArXiv.

In conclusion this paper presents the possibility of using DRL agents, specifically PPO agents, for active flow control problems that due to high-dimensionality would be too complex to solve analytically. DRL methodology based on ANNs allow efficient approximation of highly non-linear functions and can be trained through experimentation with an environment. In theory, this makes DRL methodology easily applicable to both simulations and experiments without significant changes in the methodology.

6.2 Accelerating deep reinforcement learning strategies of flow control through a multi-environment approach

From Rabault et al. [33] it became clear that the limiting factor with computations is not with the DRL methodology, but rather lies with the computational fluid dynamics part of the simulations. Significant speed-ups are required in order for the new and promising DRL methodology to be applicable to more complex flow systems.

A well-documented approach is to parallelize the numerical computations of the simulation. Another possibility is to adapt the DRL algorithm such that

the algorithm can learn from multiple independent parallel simulations.

The simulation environment and DRL algorithm are the same as for Rabault et al. [33], where a PPO algorithm is trained to control the 2-dimensional flow around a cylinder described by the Navier-Stokes equations at $Re = 100$. The flow is controlled by two small synthetic jets placed at the top and bottom of the cylinder.

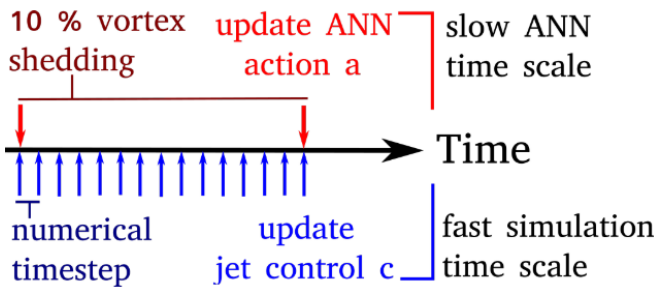


Figure 6.3: The simulations of the single cylinder system consists of two separate timescales. The numerical timestep, dt, is determined by considering the numerical stability of simulations. The slower timescale is set to capture the relevant timescale of the flow system. The action set by the DRL agent are updated according to this slower timescale. The figure corresponds to Figure 2 of [32]. Figure used from the freely available preprint at https://arxiv.org/abs/1906.10382, licensed under CC BY 4.0 at ArXiv.

Parallelizing the numerical simulations has severe limitations with regards to obtainable speed-up. In the single cylinder system the FEM problem is so small that attempts speed-ups were limited to approximately 2, independent of using more CPUs. Instead an approach where multiple independent environments feed data to the DRL agent is implemented. This means that the agent is learning by interacting with several environments at the same time. Transferring to a human context we can think of a human able to play multiple games of chess at the same time and learn from every game that is played, that is how the agent is able to interact with multiple simulations at the same time and learn from each one of them. Parallellization in way is especially well suited for DRL systems where most of the computation time is spent in the environment.

To handle the parallel stream of experiences a communication layer is added on top of the agent-environment interaction. In combination with the DRL framework of TensorForce [18] the environment wrapper allows any DRL algorithm and any environment to be parallelized. This allows the expensive numerical simulations to be distributed on multiple CPUs or on multiple computers through network communication.

Other parallelization methods based on the Hogwild method [34] have been proposed, but are mainly concerned with simulations running hundreds

of environments simultaneously where communication overhead and syncing between instances and clusters are the greatest bottlenecks. The relatively straightforward method presented by Rabault and Kuhnle [32] is better suited for moderately short learning problems where the environment is expensive to compute, like simulations of fluid dynamics.

The update period, also known as batch size, is set to 20 episodes, meaning that the policy of the DRL agent is updated after data from 20 episodes has been gathered. Using a number of parallel environments that is a divider of the update period (1, 2, 5, 10, 20) show perfect speed-up, and running simulations with synchronization (i.e. the DRL algorithm waits for all environments to finish before starting new episodes) results in what a situation that is effectively identical to training in serial.

Increasing the number of parallel environments to more than the batch size results in what can be seen as "overparallelizing" the data collection of the DRL algorithm beyond what it is naturally able to. Significant speed-up as still observed, but not quite perfect as for less environments than the batch size. The learning quality is also slightly negatively impacted with the introduction of clear steps in learning appearing. The results of overparallelization are presented in fig. 6.4. This causes the second and third policy updates to be based on off-policy data, and empirical results indicate that this does not cause problems regarding the consistency and stability of the learning algorithm.

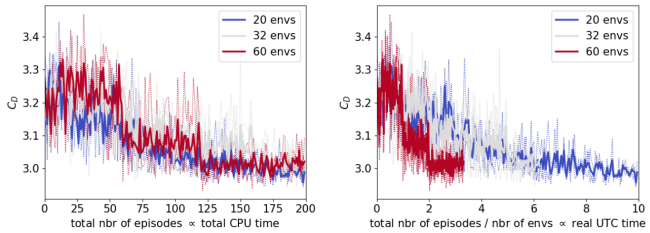


Figure 6.4: Scaling results obtained with total number of environments equal to and larger than the update period. As multiple batches finish at the same time a comparatively large cumulative improvement of the policy is observed, leading to clear steps in the learning curve. The learning speed is slightly lower when compared to training with less environments than the batch size (perfect speed-up), but significant speed-up is still present. The figure corresponds to Figure 4 of [32]. Figure used from the freely available preprint at https://arxiv.org/abs/1906.10382, licensed under CC BY 4.0 at ArXiv.

In conclusion the multienvironment approach is able to greatly speed up learning by collecting data from several independent simulations. If the number of parallel environments is more than the update period of the ANN some

of the updates are taken off-policy, but empirically this does not cause significant damage to the learning taking place. The resulting methodology is an important step towards the application of DRL methodology to more complex and realistic fluid mechanics problems.

Part II

Implementation and Methodology

Chapter 7

Technical implementation

The simulations that have been carried out in this project are quite computationally intensive and have thus been dependent on more than a personal computer to carry out the simulations. To carry out simulations we utilized cloud-based virtual machines (VMs) provided by NREC (previously known as UH-IaaS), which could be run 24/7 whenever a simulation was ready to start computations. When working with VMs in addition to multiple co-dependent python packages it can be very beneficial to utilize containerization. We chose to use Docker which is an open-source container technology initially created for Linux (which we use), but in later years also has been made available for Windows users. All code implementation was done in Python which has several well-developed libraries for ML simulations.

In this chapter we will give a brief explanation of how the VMs were set up and used together with Docker containers. We will also give an introduction to the Python packages that were used in the code implementation. The numerical flow solver will be presented in chapter 8, in addition to a selection of the code relevant for DRL simulations that will be presented in chapter 9.

7.1 NREC VMs

NREC is a collaboration between the University of Oslo and University of Bergen with additional sponsors, that provides cloud-based computational resources for academic projects. Students at both mentioned universities can apply for access to VMs for academically related projects.

In this project we have used 6 VMs of 16 virtual CPUs (vCPUs) for simulations at $Re = 100$ and one VM of 64 vCPUs for simulations at $Re = 150$. Note that a vCPU is not equal to a physical CPU core, but is rather a measure of the allocated processing time spent on a CPU. For example, a physical CPU processor with 8 physical CPU cores that each can support 8 virtual processors, can hold $8 \cdot 8 = 64$ vCPUs. Meaning that one physical CPU of 8 cores

can support 64 vCPU / 4 vCPU per VM = 16 VMs.

Each VM is initialized with Ubuntu 18.04 which is a reasonably lightweight OS. When we create the VMs we pair it with an Secure Shell (SSH) key which lets us connect to and control the VM remotely. By connecting to the VM through SSH we have full access to the OS through the terminal, and can install the software we need to run Docker containers.

7.2 Docker

A Docker container is software application that packages code and all required dependencies such that the code can be run in the same way independent of the infrastructure the container is being run on. Our container is based on a container created by the FEniCS project, which is an open-source computing platform for solving partial differential equations in Python. The fluid mechanical computations that are carried out in the project are solved using FEM code from the python package developed by FEniCS.

In the Docker container we can install all the required Python packages we need to run our simulations. We also add the GitHub repository of the project where all the code necessary for running simulations is stored. Once the Docker container is built we can simply download it to any laptop, desktop, or VM and run the code in the exact same way as on whatever machine we built the container on.

We can thus use the same initial Docker container on every VM we are using for simulations, and then do individual changes to the code on each VM in order to change the configuration of the simulations. In this way we can carry out many different simulations, all based on the same code, by adjusting a few parameters in the code on each VM.

7.3 Necessary Python packages

Before we move on to the actual code and implementation in chapter 9 we will give a brief presentation of the main Python packages that are utilized in the project. Note that a lot of packages require other packages to be installed, without them being explicitly used in the implemented code. We will list the packages that are actively used in the code, with a brief explanation of their use.

- **TensorForce:** Open-source framework for deep reinforcement learning (DRL). Based on the TensorFlow library. [18]

- **Numpy:** Numpy is probably the most used package in Python, with powerful array handling, linear algebra capabilities, basic math functions, and functions to dump data to .txt or .csv files.
- **Pandas:** Pandas is a popular package for handling data, and have in this project been used to handle data from simulations to create visualizations of the results.
- **TensorFlow:** TensorFlow is an open-source machine learning library developed by Google, and is widely used in both industry and academics. We have not used TensorFlow directly, but as a required dependency for TensorForce.
- **Matplotlib:** Matplotlib is another open-source library focused on creating visualizations. Most graphs and visualizations in this project has been made by Matplotlib.
- **FEniCS/Dolfin:** FEniCS consists of several software components which are split into separate python packages. Dolfin is the high-performance backend of FEniCS written in C++. The fluid mechanical solver utilized in this project, developed by Miroslav Kuchta, is largely written using Dolfin.
- **mpi4py:** MPI for Python provides bindings for the message passing interface (MPI) standard for Python programming, allowing Python programs to utilize multiple processors.

Chapter 8

Flow Solver

To apply flow control on the fluidic pinball system, that will be introduced further in chapter 10, we require a way to simulate the flow. The fluid flow is governed by the Navier-Stokes equations and the effects of applied control is found by computing the solution for the Navier-Stokes equations on the given domain, i.e. simulation environment. To solve the Navier-Stokes equations an incremental pressure correction scheme (IPCS) solver, Goda [10], is implemented in Python, using the `Dolfin` package of FEniCS [1]. The implementation of the solver was done by Miroslav Kuchta, and although it is possible to simply use a solver as a given tool, having some insight of the methodology and functionality can be beneficial.

We will present the code of the solver part-by-part which might cause some loss of continuity in the code presented. For those who would prefer to read the entire code as a whole we refer them to the GitHub repository of the project where all source code is available.

8.1 Python packages and domain explanation

```
1 from pinball_utils import as_mpi4py_comm, unique_points    #
    API change for MPI and function that returns unique points
    only
2
3 # Import Python packages
4 from mpi4py import MPI as pyMPI
5 from dolfin import *
6 import numpy as np
7 import sys
8
```

[1] The solver is implemented using FEniCS version 2018.1.0

```
9
10 class FlowSolver(object):
11     '''
12     Solve Navier-Stokes in the box domain below. Following
       arXiv:1812.08529
13     domains 1, 3, 4, have (Uinfty, 0) prescribed on them while
       2 is outflow
14     boundary. Boundary conditions on the cylinders (C) are
       rotations.
15
16                  4
17     ul(x)----------------ur(x)
18     |      C                |
19     | 1   C                 | 2
20     |      C                |
21     ll(x)----------------lr(x)
22                  3
23     '''
```

Listing 8.1: Flow solver - part 1 - packages and domain.

In the first section of code we import two simple functions from a separate Python file, `pinball_utils.py`. One function safeguards the MPI version dependent on the `Dolfin` version, while the other function checks that points defined in a 2D array are unique, and removes any copies if present. The other packages, `Dolfin`, `numpy`, and `mpi4py`, are presented in section 7.3. Then the **FlowSolver** class is created. The domain is also given a brief description along with boundary values, which will be implemented later in the code.

8.2 Initialization of FlowSolver attributes

```
24     def __init__(self, comm, flow_params, geometry_params,
       solver_params):
25         '''IPCS solver'''
26         mu = Constant(flow_params['mu'])   # dynamic viscosity
27         rho = Constant(flow_params['rho']) # density
28
29         mesh_file = geometry_params['mesh']
30         # Load mesh with markers
31         mesh = Mesh(comm)
32         h5 = HDF5File(comm, mesh_file, 'r')
```

```
33          h5.read(mesh, 'mesh', False)
34
35          surfaces = MeshFunction('size_t', mesh, mesh.topology
    ().dim()-1)
36          h5.read(surfaces, 'facet')
37
38          # Define function spaces
39          V = VectorFunctionSpace(mesh, 'CG', 2)
40          Q = FunctionSpace(mesh, 'CG', 1)
41
42          # Define trial and test functions
43          u, v = TrialFunction(V), TestFunction(V)
44          p, q = TrialFunction(Q), TestFunction(Q)
45
46          u_n, p_n = Function(V), Function(Q)
47          # Starting from rest or are we given the initial state
48          for path, func, name in zip(('u_init', 'p_init'), (u_n
    , p_n), ('u0', 'p0')):
49              if path in flow_params:
50                  comm = mesh.mpi_comm()
51                  XDMFFile(comm, flow_params[path]).
    read_checkpoint(func, name, 0)
52
53          u_, p_ = Function(V), Function(Q)  # Solve into these
54
55          dt = Constant(solver_params['dt'])
56          # Define expressions used in variational forms
57          U  = Constant(0.5)*(u_n + u)
58          n  = FacetNormal(mesh)
59          f  = Constant((0, 0))
60
61          epsilon = lambda u: sym(nabla_grad(u))
62          sigma = lambda u, p: 2*mu*epsilon(u) - p*Identity(2)
```

Listing 8.2: Flow solver - part 2 - Definitions and simulation setup.

In the second section of code the necessary parameters and functions are initialized. Constants like `mu` and `rho` are determined by the flow parameters that are input when calling the **FlowSolver** class in a Python script. A mesh file of `.h5` format should be provided which is read and from which the surfaces of the domain are read.

The finite element method (FEM) that is applied to solve the Navier-Stokes equations numerically involves the use of trial- and test functions. Lines 39

and 40 create a vector-valued finite element function space is created for the velocity, while a finite element function space is created for the pressure field. Note that the function spaces are defined with respect to the mesh, with elements of the Lagrange class ("CG"). The integer following "CG" in the function space calls define the degree of the finite elements. Once the function spaces are created the trial- and test functions for velocity and pressure are created. The trial- and test function for velocity are based on the same function space, and correspondingly for pressure. Line 46 represents a function of form $u_h = \sum_{i=1}^{n} U_i \phi_i$, where $\{\phi\}_{i=1}^{n}$ is a basis for a function space $\mathcal{V}_h$, in the code either function space `V` or `Q`, and U_i is the vector of degrees of freedom for the function u_h. Numerically `u_n` is the computed velocity at timestep n, u^n.

Line 48 to 51 are used to determine whether the flow parameter that are passed to the class upon creation include paths to initialized flow fields from which the simulation should be started, or whether the simulation will start from scratch.

Line 53 is essentially the same as line 46, where `u_` is the latest computed approximation of the velocity, u^{n+1}, while `u` is the unknown velocity at the next timestep, mathematically given as u^{+1}. The same naming convention is applied to the pressure. Lines 57 to 62 define different expressions that are used when expressing the problem in variational form.

8.3 Variational form and boundary conditions

```
64          # Define variational problem for step 1
65          F1 = (rho*dot((u - u_n) / dt, v)*dx
66                + rho*dot(dot(u_n, nabla_grad(u_n)), v)*dx
67                + inner(sigma(U, p_n), epsilon(v))*dx
68                + dot(p_n*n, v)*ds-dot(mu*nabla_grad(U)*n, v)*ds
69                - dot(f, v)*dx)
70
71          a1, L1 = lhs(F1), rhs(F1)
72
73          # Define variational problem for step 2
74          a2 = dot(nabla_grad(p), nabla_grad(q))*dx
75          L2 = dot(nabla_grad(p_n), nabla_grad(q))*dx - (1/dt)*
    div(u_)*q*dx
76
77          # Define variational problem for step 3
78          a3 = dot(u, v)*dx
```

```
79          L3 = dot(u_, v)*dx - dt*dot(nabla_grad(p_ - p_n), v)*
    dx
80
81          # Same inflow profile as Noack. (1, 0) unit vector
82          U_infty = Constant((flow_params['U_infty'], 0))  #
    Scaled e_x
83          # Define boundary conditions for non-cylinder
    boundaries
84          bcu_inlet = DirichletBC(V, U_infty, surfaces, 1)
85          bcu_top = DirichletBC(V, U_infty, surfaces, 4)
86          bcu_bot = DirichletBC(V, U_infty, surfaces, 3)
87          # Fixing outflow pressure
88          bcp_outflow = DirichletBC(Q, Constant(0), surfaces, 2)
```

Listing 8.3: Flow solver - part 3 - IPCS scheme and boundary conditions.

When using an IPCS solver we need to define three variational problems, one for each step in the IPCS scheme. The IPCS schemes three steps are, first compute the tentative velocity by solving for `u` in variational problem step 1, then compute the corrected pressure by solving for `p` in variational problem step 2, and finally compute the corrected velocity by solving variational problem step 3 [40].

In line 82 the inflow profile is defined as a vector in the x-direction, where the inflow velocity, `U_infty` is equal to 1 in our simulations, thus defining the inflow as the unit vector in the x-direction. Following the inflow definition the boundary conditions of the surrounding domain are set, note the exception of the boundary conditions for the cylinders which are determined by a function call later in the code.

For a complete introduction to the implemention of PDE solvers, including Naiver-Stokes solvers, with FEniCS we refer to the freely accessible FEniCS tutorial by Langtangen and Logg [19].

8.4 Setting up cylinders and matrices for solutions

```
89          # Finally we have rotations on the cylinders
90          tags, exprs, info = FlowSolver.setup_cylinder_bcs(
    surfaces, 4)
91          bcu_cylinder = [DirichletBC(V, expr, surfaces, tag)
    for tag, expr in zip(tags, exprs)]
92
93          # All bcs objects together
```

```
94          bcu = [bcu_inlet, bcu_top, bcu_bot] + bcu_cylinder
95          bcp = [bcp_outflow]
96
97          As = [Matrix(comm) for i in range(3)]
98          bs = [Vector(comm) for i in range(3)]
99
100         # Assemble matrices
101         assemblers = [SystemAssembler(a1, L1, bcu),
102                       SystemAssembler(a2, L2, bcp),
103                       SystemAssembler(a3, L3, bcu)]
104
105         # Apply bcs to matrices (this is done once)
106         for a, A in zip(assemblers, As):
107             a.assemble(A)
108
109         # Chose between direct and iterative solvers
110         solvers = [LUSolver(comm, A, 'mumps') for A in As]
111         # Set matrices for once, likewise solver don't change
    in time
112
113         gtime = 0.  # External clock
```

Listing 8.4: Flow solver - part 4 - Cylinders and matrix setup.

Setting up the cylinders, i.e. locating positions and determining the boundary conditions for each is done by the separate function `setup_cylinder_bcs`. We will come back to the functionality of this function in section 8.7.

The boundary conditions of the cylinders and the rest of the computational domain are combined such that boundary conditions of velocity are defined by `bcu` and boundary conditions for pressure are defined by `bcp`. As the variational steps are time-independent we can assemble the matrices and vectors manually instead of automatically through FEniCS's `solve()` function which lets us assemble them once and for all outside the time-stepping loop.

8.5 Making attributes of the FlowSolver class accessible

```
114         # Things to remember for evolution
115         self.cylinder_bc_exprs = exprs
116         self.cylinder_bc_tags = tags
117         self.cylinder_info = info
```

```
118         # Keep track of time so that we can query it outside
119         self.gtime, self.dt = gtime, dt
120 
121         self.solvers = solvers
122         self.assemblers = assemblers
123         self.bs = bs
124         self.u_, self.u_n = u_, u_n
125         self.p_, self.p_n= p_, p_n
126 
127         # Rename u_, p_ for to standard names (simplifies
    processing)
128         u_.rename('velocity', '0')
129         p_.rename('pressure', '0')
130 
131         tags = tuple(map(int, tags))
132         # Also expose measure for assembly of outputs outside
133         self.ext_surface_measures = [Measure('ds', domain=mesh
    , subdomain_data=surfaces, subdomain_id=tag)
134                                      for tag in tags]
135 
136         self.viscosity = mu
137         self.density = rho
138         self.normal = n
139         # Finally the communicator
140         self.comm = as_mpi4py_comm(comm)
```

Listing 8.5: Flow solver - part 5 - Make FlowSolver attributes accessible.

This section of code lets us access velocity, pressure, and boundary conditions outside the `__init__` function of the **FlowSolver** class.

8.6 Evolving the flow and applying rotations

```
141     def evolve(self, bc_values):
142         '''Make one time step with the given rotation
    magnitudes'''
143         assert len(bc_values) == len(self.cylinder_bc_tags), (
    bc_values, self.cylinder_bc_tags)
144         # Set rotation
145         for expr, value in zip(self.cylinder_bc_exprs,
    bc_values):
```

```
146             expr.A = value
147
148         # Make a step
149         self.gtime += self.dt(0)
150
151         assemblers, solvers = self.assemblers, self.solvers
152         bs = self.bs
153         u_, p_ = self.u_, self.p_
154         u_n, p_n = self.u_n, self.p_n
155
156         solution_okay = True
157         for (assembler, b, solver, uh) in zip(assemblers, bs,
    solvers, (u_, p_, u_)):
158             assembler.assemble(b)
159             try:
160                 solver.solve(uh.vector(), b)
161             except:
162                 solution_okay = False
163
164         solution_okay = solution_okay and not np.any(np.isnan(
    u_.vector().get_local()))
165         solution_okay = solution_okay and not np.any(np.isnan(
    p_.vector().get_local()))
166         # Reduce accross CPUs
167         solution_okay = self.comm.allreduce(solution_okay, op=
    pyMPI.PROD)
168
169         if not solution_okay:
170             print('Simulation gone wrong')
171             sys.exit()
172
173         u_n.assign(u_)
174         p_n.assign(p_)
175
176         # Share with the world
177         return u_, p_, solution_okay
```

Listing 8.6: Flow solver - part 6 - Evolving the flow.

`evolve()` is the main function of the **FlowSolver** class which computes the flow as it develops. The function accepts boundary conditions for the cylinders as input. Any non-zero boundary condition on a cylinder will correspond to a rotation that will affect the flow. Calling **FlowSolver**`.evolve()`

in a script where the **FlowSolver** class has been initialized will compute one numerical timestep with given rotations of the cylinders.

In line 145 the rotational magnitude defined by the boundary conditions accepted as input are applied to each cylinder before computing the velocity and pressure fields. The actual computations are done in lines 157 to 163 where the script also checks that no undefined values for velocity or pressure appears in the simulation. Line 167 collects the results of distributed computations in the case of MPI processing, and returns a flag for whether the simulation ended without undefined values. If a simulation returns undefined values the simulation will stop with an error message, and we will have to debug the scripts where the solver is called and consider the physics and mesh quality of the simulation we want to compute.

8.7 Cylinder setup with boundary conditions for rotations

```
178     @staticmethod
179     def setup_cylinder_bcs(surfaces, tag):
180         '''Discover cylinders and make rotation expression for
        them'''
181         # By convention cylinders are labels after tag; local
182         tags = [t for t in set(surfaces.array()) if t > tag]
183 
184         mesh = surfaces.mesh()
185         comm = as_mpi4py_comm(mesh.mpi_comm())
186         # Global tags are
187         tags = list(set(sum(comm.allgather(list(tags)), [])))
188 
189         x = mesh.coordinates()
190 
191         # On each surface the value is given by customizing
        the following template
192         rot_expr = lambda: Expression(('A*(x[1]-CY)/sqrt((x[0]
        -CX)*(x[0]-CX) + (x[1]-CY)*(x[1]-CY))',
193                                        '-A*(x[0]-CX)/sqrt((x
        [0]-CX)*(x[0]-CX) + (x[1]-CY)*(x[1]-CY))'),
194                                       degree=1, CX=0, CY=0, A
        =0)
195 
196         mesh.init(1, 0)
```

```
197         values, cylinder_info = [], []
198         for tag in tags:
199             # Discover center points as center of mass of
    vertices lying
200             # on the cylinder
201             v_idx = sum((list(f.entities(0)) for f in
    SubsetIterator(surfaces, tag)), [])
202             # Send the points to root
203             global_circle_points = comm.gather(x[v_idx], 0)
204 
205             center_x, center_y, radius = (None, )*3
206             # Let it compute the info
207             if comm.rank == 0:
208                 global_circle_points = unique_points(np.
    row_stack(global_circle_points))
209                 center_x, center_y = np.mean(
    global_circle_points, axis=0)
210                 radius = np.linalg.norm(global_circle_points
    [0] - np.array([center_x, center_y]))
211 
212                 comm.bcast((center_x, center_y, radius), 0)
213             # Just listen
214             else:
215                 center_x, center_y, radius = comm.bcast((
    center_x, center_y, radius), 0)
216 
217             expr = rot_expr()
218             expr.CX = center_x
219             expr.CY = center_y  # Leaving magnitude to the
    controller
220 
221             values.append(expr)
222             cylinder_info.append((center_x, center_y, radius))
223         return tags, values, cylinder_info
```

Listing 8.7: Flow solver - part 7 - Setting up cylinders and make ready for rotations.

The final function of the **FlowSolver** class that is necessary for our simulations takes the surfaces of the domain, i.e. our three cylinders and a tag for each as input. In the end the function outputs the rotation of each cylinder, along with information of the cylinders positions, and a tag. After finding each cylinder they are given a rotation defined by the expression in line 192.

The root process calculates the position of each cylinder and shares it with each processor through `.bcast`, while the other processes uses `.bcast` to receive the results from the root process.

The cylinder position is then applied to the expression defining rotation of the cylinders, while the final values that determine the magnitude of rotation, `A`, is left to be defined in **FlowSolver**`.evolve()`.

Chapter 9

Code implementation

In chapter 8 we presented the strictly necessary fluid mechanical part of the simulations. In addition to the flow solver we need several Python scripts in order to run control the simulation using deep reinforcement learning agents. The main class is a custom **TensorForce** environment consisting of almost 900 lines of code[1]. A good amount of those 900 lines are not strictly necessary for running a simulation, but include dump routines for drag and lift values in `.csv` format, and the possibility of dumping **.pvd** files of the pressure and velocity fields for visual inspection in ParaView. To start a simulation we also need to configure the DRL agent and determine how long the agent should be allowed to train for. This is done in the script called `launch_parallel_training.py` and we will give an overview of the most important snippets. For further explanations of the code and implementation we refer to the provided READMEs that can be found in the GitHub repository.

9.1 TensorForce environment class

The base class of a customizable TensorForce environment (version 0.5.0) can be found in the TensorForce GitHub repository. Note that the custom environment implemented for the fluidic pinball simulations differ quite significantly from the custom environment template, and we will only present the essential functions of the class here. Thus, significant amounts of the source code is left out and the code presented here is not meant to be a working example. For the complete code we refer to the source file available in the thesis repository where the complete source code is available.

```
class Env2DPinball(Environment):
```

[1]Code developed for TensorForce 0.5.0 - Major updates to the library has been added since.

```
    """Environment Class for 2D flow simulation of the fluidic
     pinball."""

    def __init__(self, path_root, geometry_params, flow_params
, solver_params, output_params,
                 optimization_params, inspection_params,
n_iter_make_ready=None, verbose=0, size_history=2000,
                 reward_function='plain_drag_lift',
size_time_state=50, duration_execute=0.5, simu_name="Simu",
     root_folder='mesh/re100'):
        """
        """

        # Make the input parameters available to the rest of
the class
        self.path_root = path_root
        self.root_folder = root_folder

        self.flow_params = flow_params
        self.geometry_params = geometry_params
        self.solver_params = solver_params

        self.comm = mpi_comm_world()

        # Initialize drag, lift and recirc area for each
cylinder.
        self.episode_drags0 = np.array([])

        self.start_class(complete_reset=True)
```

Listing 9.1: TensorForce environment class - __init__

The first section of code we present is the `__init__` method, known as a *constructor*, and is called when an object is created from the environment class. The method allows the class to initialize the attributes of the class, and make them available to the rest of the class by redefining the input parameters of the `__init__` method as `self.input_parameter`.

Parameters defining the flow are passed to `flow_params`, `geometry_params` define geometrical values like cylinder locations, and `solver_params` contains the numerical timestep that is to be used during a simulation. These values are all defined in the script `simulation_base/env.py`, which is used to configure each individual simulation. Finally we initialize arrays where we want to save drag, lift, reward, and other variables that are relevant to the simu-

lation, and call the `start_class` method which will start the first episode of simulations.

```
    def start_class(self, complete_reset=True):
        if complete_reset == False:
            self.solver_step = 0
        else:
            self.solver_step = 0
            self.accumulated_drag = 0

            # Dictionary to store simulation data
            self.history_parameters = {}

            # Set Path to .msh and .h5 file.
            msh_file = '.'.join([self.path_root, 'msh'])
            h5_file = '.'.join([self.path_root, 'h5'])

            if not os.path.exists(h5_file):
                # if no .h5 file of mesh, convert .msh to .h5
                mesh = convert(msh_file, h5_file)

            # if necessary, load initialization fields
            if self.n_iter_make_ready is None:
                self.flow_params['u_init'] = '/'.join([self.
root_folder, 'u_init.xdmf'])
                self.flow_params['p_init'] = '/'.join([self.
root_folder, 'p_init.xdmf'])

            # Create flow simulation object
            self.flow = FlowSolver(self.comm, self.flow_params
, self.geometry_params, self.solver_params)

            # Setup probes
            if self.output_params["probe_type"] == 'pressure':
                self.ann_probes = PressureProbe(self.flow,
self.output_params['locations'])

            # probe setup for Pinball solver
            self.drag_probes = [DragProbe(i, self.flow) for i
in range(len(self.geometry_params['cylinder_center']))]

            # Initialize rotation and action as zeros
```

```
            self.Qs = np.zeros(num_cylinders)
            self.action = np.zeros(num_cylinders)

            # if necessary, create initialized fields
            if self.n_iter_make_ready is not None:
                self.u_, self.p_, self.status = self.flow.
    evolve(self.Qs)

                for _ in range(self.n_iter_make_ready):
                    self.u_, self.p_, self.status = self.flow.
    evolve(self.Qs)
                    self.probes_values = self.ann_probes.
    sample(self.u_, self.p_).flatten()
                    self.drag = [dp.sample(self.u_, self.p_)
    for dp in self.drag_probes]
                    self.solver_step += 1

            if self.n_iter_make_ready is not None:
                encoding = XDMFFile.Encoding.HDF5
                mesh = convert(msh_file, h5_file)
                comm = mesh.mpi_comm()

                u_init = '/'.join([self.root_folder, 'u_init.
    xdmf'])
                p_init = '/'.join([self.root_folder, 'p_init.
    xdmf'])

                # save field data
                XDMFFile(comm, u_init).write_checkpoint(self.
    u_, 'u0', 0, encoding)
                XDMFFile(comm, p_init).write_checkpoint(self.
    p_, 'p0', 0, encoding)

                sys.exit("\nInitialization fields have been
    created!\nReset simulation using make_converge=False\n")

            self.ready_to_use = True
```

Listing 9.2: TensorForce environment class - start_class

The `start_class` method is used to start a new episode of simulations, with the added possibility of resetting the simulation completely, i.e. `complete_reset = True`. If the simulation is starting from scratch the method checks for a `.h5`

file of the computational mesh which is needed for the **FlowSolver** class to carry out the numerical computations of the Navier-Stokes equations. If no `.h5` file is found the external function `convert`, imported from a separate script, converts a `.msh` file to the `.h5` format. The path to converged initialization fields are stored in `flow_params`, if available, before the **FlowSolver** class presented in chapter 8 is called and we create the flow solver object `self.flow`.

A variety of probes are then set up, e.g. drag probes computing the drag on each cylinder, and pressure probes that are used to sample the flow and represents the state given to the DRL agent. Note that the agent is not given a complete state of the system, but a quite sparse representation of the flow (See fig. 10.8).

The control actuations are initialized as zero-rotation, which is necessary if the script is used to create a converged initialization fields of the pressure and velocity by simulating the flow without actuations for a long time. The convergence of initialization fields is explained in more detail in chapter 10. If converged initialization fields are created they are saved as `.xdmf` files which will then be loaded in at the start of each episode.

```
    def states(self):
        """
        Returns:
            States specification, with the following
    attributes
                (required):
                - type: 'float'
                - shape: integer, or list/tuple of integers (
    required).
        """
        if self.output_params["probe_type"] == 'pressure':
            return dict(type='float',
                        shape=(len(self.output_params["
    locations"]) * \
                               self.optimization_params["
    num_steps_in_pressure_history"],)
                        )
        elif self.output_params["probe_type"] == 'velocity':
            return dict(type='float',
                        shape=(2 * len(self.output_params["
    locations"]) * \
                               self.optimization_params["
    num_steps_in_pressure_history"],)
```

```
                )
```

Listing 9.3: TensorForce environment class - states

The `states` method defines how the state of the system is represented to the DRL agent. The exact method of sampling is determined `start_class`, but the agent requires a formal specification of the size of each state.

```
    def actions(self):
        """
        Returns:
            actions (spec, or dict of specs): Actions
    specification, with the following attributes
                (required):
                - type: 'float' (required).
                - shape: list/tuple of integers (default: []).
                - min_value and max_value: float
        """
        return dict(type='float',
                    shape=(len(self.geometry_params["
    cylinder_center"]), ),
                    min_value=self.optimization_params["
    min_rotation_cyl"],
                    max_value=self.optimization_params["
    max_rotation_cyl"]
                    )

    def close(self):
        """
        Close environment. No other method calls possible
    afterwards.
        """
        self.ready_to_use = False
```

Listing 9.4: TensorForce environment class - actions and close

Similarly the `states` method, the `actions` method formally defines the action space of the agent, i.e. how many actuations should be returned and what values are allowed for each actuation. In our case of the fluidic pinball we have three cylinders which require 1 actuation value each and each actuation must be in the interval $[-1, 1]$.

The `close` method simply closes the environment at the end of simulations.

```
    def reset(self):
        """
        Reset environment and setup for new episode.
        Returns:
            initial state of reset environment.
        """
        if self.solver_step > 0 and not self.flag_need_reset:
            mean_accumulated_drag = self.accumulated_drag /
    self.solver_step
            mean_accumulated_lift = self.accumulated_lift /
    self.solver_step

            if self.verbose > -1:
                print("mean accumulated drag on the whole
    episode: {}".format(mean_accumulated_drag))

        chance = random.random()

        probability_hard_reset = 0.2

        # 20% chance for a complete reset
        if chance < probability_hard_reset or self.
    flag_need_reset:
            self.start_class(complete_reset=True)
            self.flag_need_reset = False
        else:
            self.start_class(complete_reset=False)

        next_state = np.transpose(np.array(self.probes_values)
    )
        if self.verbose > 0:
            print(next_state)

        self.episode_number += 1

        return(next_state)
```

Listing 9.5: TensorForce environment class - reset

The `reset` method calculates the accumulated drag of an episode, resets the episode by calling `start_class`, and increases the episode counter by 1.

```
    def execute(self, actions):
        """
        Executes action, observes next state(s) and reward.
        Args:
            actions: Actions to execute.
        Returns:
            Tuple of (next state, bool indicating terminal,
    reward)
        """
        try:
            action = actions

            if action is None:
                # No rotation given, set rotation as zero
                action = np.zeros((num_cylinders, ))

            self.previous_action = self.action
            self.action = action

            # To execute several numerical integration steps
            self.last_actuation = 0
            self.time = 0

            # Resets between every new action given to execute
    ()
            self.current_numerical_step = 0

            while (self.time - self.last_actuation) < self.
    duration_execute:
                self.current_dt = self.solver_params['dt']
                self.time += self.current_dt

                if "smooth_control" in self.
    optimization_params:
                    # Apply smoothing to avoid sudden changes
    in rotation.
                    self.Qs += self.optimization_params["
    smooth_control"] * (np.array(action) - self.Qs)

                # Evolve the flow one numerical timestep with
    rotations Qs
                self.u_, self.p_, self.status = self.flow.
```

```
    evolve(self.Qs)

                # Solver step resets every episode, current
    resets every action.
                self.solver_step += 1
                self.current_numerical_step += 1

                # sample probes, drag, and lift
                self.probes_values = self.ann_probes.sample(
    self.u_, self.p_).flatten()
                self.drag = [dp.sample(self.u_, self.p_) for
    dp in self.drag_probes]
                self.lift = [lp.sample(self.u_, self.p_) for
    lp in self.lift_probes]

                # write sampled data to the history buffers
                self.write_history_parameters()

                self.accumulated_drag += np.mean(self.drag)

            self.last_actuation = self.time

            next_state = np.transpose(np.array(self.
    probes_values))

            terminal = False
            reward = self.compute_reward()

        except:
            # If exception, something has gone wrong.
            # Call functions that check state and reward for
    NaN or Inf values
            # and reset simulation if invalid values found.
            terminal = True

        return (next_state, terminal, reward)
```

Listing 9.6: TensorForce environment class - execute

The final method we present is where the actual simulation takes place. The `execute` method takes a list of actions as input, if no actions are given we default to no rotation, and carries out the applied actions given. During a baseline simulation the actuations are zero, while if we are training or evalu-

ating a DRL agent the actuations are given by the agent. `duration_execute` is calculated in `simulation_base/env.py` and determines how many timeunits each set of actuations should be applied. We also apply smoothing of the actuations to avoid sudden changes in rotation leading to infinite acceleration which can break the physics of the simulation. Thus, each new actuation, Q^{n+1} is given as $Q^{n+1} = Q^n + \alpha(a - Q^n)$, where Q^n is the previously applied actuation, a is the actions provided by the DRL agent, and α is a smoothing parameter.

Once the actuations are determined we pass the array of actuations to the `evolve` method of the **FlowSolver** class which computes the flow for the next numerical timestep. The drag, lift, and pressure probes are then sampled and values stored by the `write_history_parameters` method (not described here).

Once the `duration_execute` loop is finished with one given set of actuations from the DRL agent the next observed state is prepared and the reward corresponding to the given set of actuations is calculated. Some simulations have been observed to break which is why we include `try-except` blocks. If the simulation breaks we call a method that checks the state and the reward for infinite or undefined values. If the simulation breaks we return `terminal = True` which will reset and start a completely new episode.

9.2 TensorForce agent and simulation start

Once the custom environment class is properly defined with all necessary methods we need a script to start the simulations. As for the environment script we will not present everything here, but focus our attention on the configurable parts of the script.

At the start of the script a number of environment objects, all named `example_environment`, of the `Env2DPinball` class are created, one environment per parallel process. These environments are stored in a list of environments simply named `environments`, while one single environment is used for evaluation and is named `evaluation_environment`.

```
network = [dict(type='dense', size=512), dict(type='dense',
    size=512)]

agent = Agent.create(
    # Agent + Environment
    agent='ppo', environment=example_environment,
    max_episode_timesteps=nb_actuations,
    # Network
    network=network,
```

```
    # Optimization
    batch_size=20, learning_rate=1e-3, subsampling_fraction
   =0.2, optimization_steps=25,
    # Reward estimation
    likelihood_ratio_clipping=0.2, estimate_terminal=True,
    # Critic
    critic_network=network,
    critic_optimizer=dict(
        type='multi_step', num_steps=5,
        optimizer=dict(type='adam', learning_rate=1e-3)
    ),
    # Regularization
    entropy_regularization=0.01,
    # TensorFlow etc
    parallel_interactions=number_servers,
    saver=dict(directory=os.path.join(os.getcwd(), 'saver_data
   '), seconds=72000),
)
```

Listing 9.7: Define agent and neural network

After the necessary environments have been created we define a dense neural network that defines the policy function estimator, i.e. the *network* parameter. We also have a neural network that is trained to approximate the state value function, $V(s)$, used to calculate the *advantage* function, $\hat{A}_t$ of section 4.4, i.e. the *critic_network* parameter. For details on the state value function and advantage function estimation we refer to Schulman et al. [37]. The neural networks are implemented with ReLU activation functions, as described in subsection 3.3.2.

We then call the `create` method of the **Agent** class provided from TensorForce, which will create a DRL agent from a given specification. We specify that we want a PPO agent, then pass a single example of the custom environment the agent should interact with, and the maximum number of agent actuations per episode.

Then follows a variety of hyperparameters like the batch size (number of episodes per update batch), learning rate, and the clipping parameter, `likelihood_ratio_clipping`, corresponding to the clipping parameter of Schulman et al. [37] that gave the best results in their experiments with the PPO algorithm.

Finally we specify how many parallel environments are running, and specify how to save the agent during training.

```
runner = ParallelRunner(
```

```
    agent=agent, environments=environments,
    evaluation_environment=evaluation_environment
)

runner.run(
    num_episodes=800, max_episode_timesteps=nb_actuations,
    sync_episodes=True,
    evaluation_callback=evaluation_callback_2,
    save_best_agent=use_best_model
)
runner.close()
```

Listing 9.8: Define runner and start simulation

Once the agent has been created we initialize the runner utility where we pass the created agent, the parallel environments, and the evaluation environment.

Using the `run` method of the **ParallelRunner** class we define how many episodes the agent should train for. `sync_episodes=True` as long as the number of parallel processes is less than the batch size, but for more complex systems where more parallel processes are needed this should be checked as `False` to avoid some environments waiting on the rest of the environments to finish before starting the next episode.

Once the training is complete the runner is closed, and the trained agent is saved. When an agent has finished training we want to evaluate the agents performance. During training each updated action is determined by a probability distribution of which action is supposedly good for the given state of the flow. When we evaluate an agent we use so-called deterministic evaluation, i.e. the action to be taken for every state is the action that is most likely to maximize the accumulated reward, according to the probability distribution describing all possible actions. During training each action the chosen action is not necessarily the supposedly best one, and this introduces a natural form of exploration noise. The script to run the evaluation is called `single_runner.py`, and is in essence quite similar to `launch_parallel_training.py`. The main differences are outlined below:

```
agent = Agent.create(
    # Agent + Environment
    agent='ppo', environment=example_environment,
    max_episode_timesteps=nb_actuations,
    # Network
    network=network,
    # Optimization
```

```
    batch_size=20, learning_rate=1e-3, subsampling_fraction
   =0.2, optimization_steps=25,
    # Reward estimation
    likelihood_ratio_clipping=0.2, estimate_terminal=True,
    # Critic
    critic_network=network,
    critic_optimizer=dict(
        type='multi_step', num_steps=5,
        optimizer=dict(type='adam', learning_rate=1e-3)
    ),
    # Regularization
    entropy_regularization=0.01,
    # TensorFlow etc
    parallel_interactions=1,
    saver=saver_restore, # path to saved agent
)

agent.initialize()
###################################

def one_run():
    print("start simulation")
    state = example_environment.reset()
    example_environment.render = True

    for k in range(3*nb_actuations):
        action = agent.act(state, deterministic=deterministic,
    independent=True)
        state, terminal, reward = example_environment.execute(
   action)
```

Listing 9.9: Single runner evaluation

The agent specification is close to identical of the specification for starting training, except that instead of saving the agent we load the trained agent from the folder it was saved to during training. Once the agent is initialized with `agent.initialize()` we define the function that runs the evaluation simulation. We reset the state of environment before we start the loop that will run the simulation. The evaluation is run for three times the length of a training episode (`3*nb_actuations`) where the actions are given by the `agent.act()` call before being passed to the `execute` method of the environment class. Note that instead of determining actions by calling `agent.act()` we can also define actions as constant or following a simple function, e.g.

`action = [0.5, 0.5, 0.5]` which would correspond to constant rotation on all three cylinders. We could also define actions as `action = [sin(2k), sin(k), sin(0.5k)]`, corresponding to sinusoidal control as a function of actuation number.

9.3 Advantages of the implementation

The DRL training is parallelized with almost perfect speed-up with the same methodology as used in Rabault and Kuhnle [32]. The agent is trained by gathering experience from multiple parallel environments, allowing us to study more computationally intensive problems.

Finding converged initialization fields can be done in two different ways. In the **Env2DPinball** class it is done in serial, which can take very long on a large computational mesh which is most likely needed on more complex problems. The script found in `converge_flow/create_init_fields.py` takes advantage of MPI to compute converged initialization fields where tests have shown a reduction in computing time of $> 50\%$ for 3 and more CPUs compared to the same simulation running in serial. Note that the speed-up of the MPI convergence script does not scale linearly with the amount of processors, but speed-up is very prominent when increasing from 1 to 4 processors. Speed-up was observed to be noticeable up to 8 processors, but with decreasing difference between 5-8 processors. The recommended number of parallel processors for the MPI script will depend on the computer hardware running the simulation.

9.4 Possible improvements

The code is based on an older version of the TensorForce library, and documentation of the older code is not as extensive as the more recent versions. Further development of the code implemented in this project could be more of a challenge because of this, but the same functionality is still present in the latest TensorForce version, including the parallel training of DRL agents which is now functionally built into the TensorForce framework, rather than being handled by separate scripts.

The single runner evaluation is run in serial, which makes evaluation of an agent slower than what it could be if MPI was successfully implemented in such a way that the final evaluation could be run using parallel processing. During training it is more efficient to use the already implemented strategy of using 1 CPU per environment, which is why MPI implementation in the code where we use DRL agents has not been a priority.

Chapter 10

Methodology - Fluidic Pinball

10.1 Simulation Environment

The simulation environment used for the fluidic pinball is based on the descriptions given in Deng et al. [5]. The flow of the system is governed by the Navier-Stokes equations in the computational domain, and are solved numerically using the FEM method implemented in the FEniCS framework [21]. As given by Deng et al. [5] the Cartesian origin is placed in the middle of the two trailing cylinders. The three cylinders form an equilateral triangle of sides $3R$. In Cartesian coordinates the computational domain is given by $[-6, 20] \times [-6, 6]$, as shown in 10.1. The boundary condition on the cylinders are no-slip conditions, $U_r = 0$. On the inflow, upper, and lower boundaries a far field boundary condition is applied, $U_\infty = \mathbf{e}_x$. Where $\mathbf{e}_x$ is the unit velocity in the x-direction, given as a vector, $\mathbf{e}_x = (1, 0)$. The outflow boundary is assumed to be stress-free.

The Reynolds number (Re) based on the diameter of a single cylinder is defined as $Re = UD/\mu\nu$, where U is the inflow velocity, D is the diameter of a single cylinder, and ν is the kinematic viscosity of the flow. For all simulations we have $U = \mathbf{e}_x = (1, 0)$ and $D = 1$, while we change the value of ν to achieve the desired Reynolds number. For $Re = 100$ simulations the kinematic viscosity is thus chosen as $\nu = 0.01$, for $Re = 150$ $\nu = 0.00667$, and for the experimental $Re = 200$ simulation we have $\nu = 0.005$.

10.1.1 Mesh creation

The mesh was created using the gmsh software [9]. The mesh is split into three regions of different refinement levels. The area closest to the cylinders is where we expect the most challenging physics to happen, and it's therefore necessary to refine the mesh of this area the most. Behind the cylinders an unsteady periodic wake develops which will be affected when we apply

rotation to the cylinders. Thus, we refine the area immediately behind the cylinders more than the outlying regions of the system, but it's not necessary with as high levels of refinement as directly around the cylinders. The rest of the mesh is kept relatively coarse as the flow in these regions will not experience rapid changes in the same way as closer to the rotating cylinders.

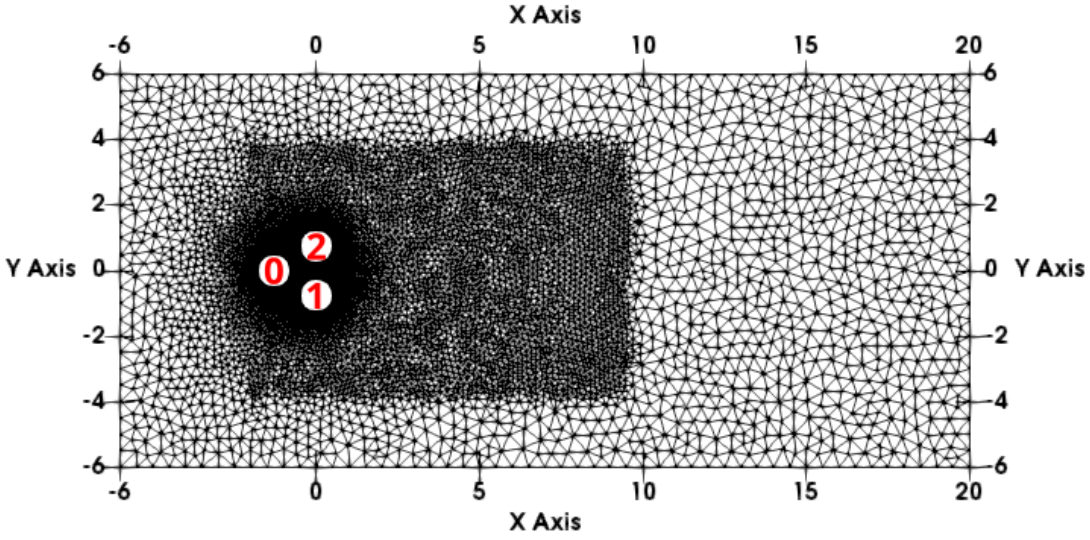


Figure 10.1: The mesh used in flow control simulations at $Re = 100$. For the mesh convergence study, and for simulations at higher Re the mesh is refined for the entire domain following the same refinement regions as can be seen in the figure.

10.2 Mesh Refinement Study

To be certain the simulations are done correctly without the computational mesh influencing the results in any significant way we simulate the system using several different meshes of different refinement levels. If there are significant differences between the meshes further refinement would have to be done, until the difference between the meshes is satisfactory small.

10.2.1 Mesh refinement at $Re = 100$

For simulations at $Re = 100$ we compared 4 different meshes of the following specifications.

Using the MPI parallelized script for computing the converged flow, based on the flow solver and a test script developed by Miroslav Kuchta, we simulate the flow for a very long time until the flow is converged on the different

Mesh name	# of cells
Coarser	16930
Simulation	26480
Refined 1	43806
Refined 2	76552

Table 10.1: Refinement levels of convergence study at $Re = 100$. Using numerical time steps of size $dt = 0.005$.

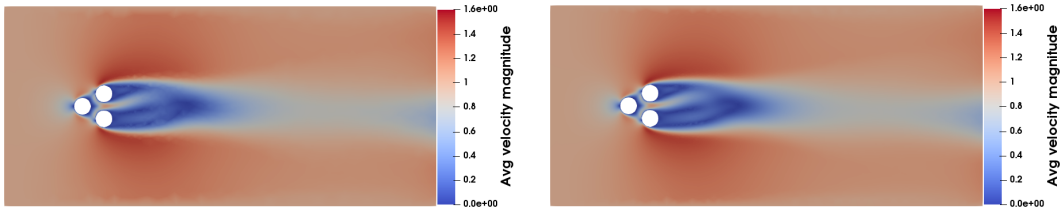
meshes. Each simulation is done without any control applied to the cylinders, and the flow is allowed to develop naturally. After approximately 800 non-dimensional time units the flows reach an unsteady periodic state with vortex shedding trailing the cylinders. To make sure that each simulation has stabilized into its respective periodic regime we continue simulating until reaching 1200 non-dimensional time units. We then compare the average drag- and lift coefficient of each mesh configuration . We calculate the mean-sum-drag of the 3 cylinders. That is, we sum the drag coefficients of the three cylinders, then take the mean of that sum over the last 100 time units of the simulation, i.e. long after the flow is fully developed. The calculation will look something like this:

$$C_D = \frac{\sum_{t=1100}^{1200} \left(\sum_{i=0}^{2} C_{D_i}^t \right)}{\Delta T}, \tag{10.1}$$

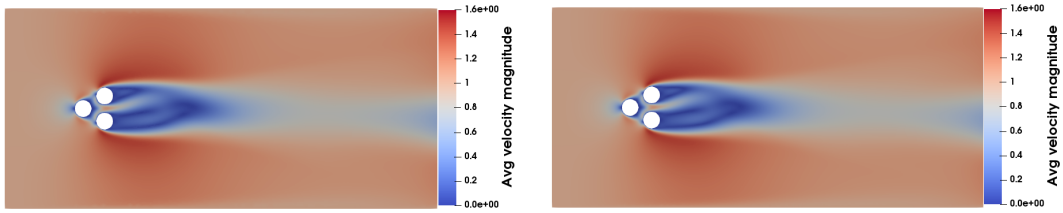
where $\Delta T = t_2 - t_1 = 1200 - 1100$, i.e. the last 100 time steps, and i indicates which cylinders drag value we add to the sum at the given time step t.

Mesh name	$C_D \pm$ std	Deviation C_D	$C_L \pm$ std	Deviation C_L	# of cells
Coarser	3.8420 ± 0.0025	0.0651 %	-0.0542 ± 0.0291	53.69 %	16930
Default	3.8405 ± 0.0028	0.0729 %	-0.0535 ± 0.0292	54.58 %	26480
Refined 1	3.8402 ± 0.0029	0.0755 %	-0.0541 ± 0.0289	53.42 %	43806
Refined 2	3.8403 ± 0.0029	0.0755 %	-0.0537 ± 0.0291	54.19 %	76552

Table 10.2: Drag and lift results as sum of all 3 cylinders for the last 100 time units of baseline simulations. The mean coefficient values with corresponding standard deviations are calculated on the interval from time 1100 to 1200 (last 100 time units). Deviation is calculated as $C_{D/L} = (1\,\text{std}/\text{mean}\, C_{D/L}) * 100$. E.g. 1 standard deviation / mean $= (0.0028/3.8405) * 100 = 0.0729$ %. I.e. 1 standard deviation equals 0.0729 % of the mean C_D for the Default mesh. As the lift coefficient of the baseline simulation is close to zero-centered the standard deviation of the lift coefficient corresponds to a large percentage of the mean lift coefficient.

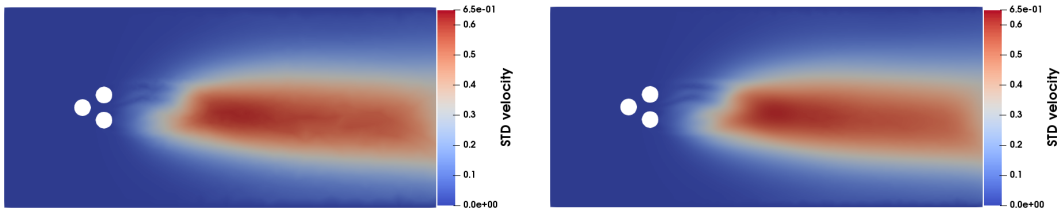


(a) Coarser mesh (16930 cells). **(b)** Simulation mesh (26480 cells).

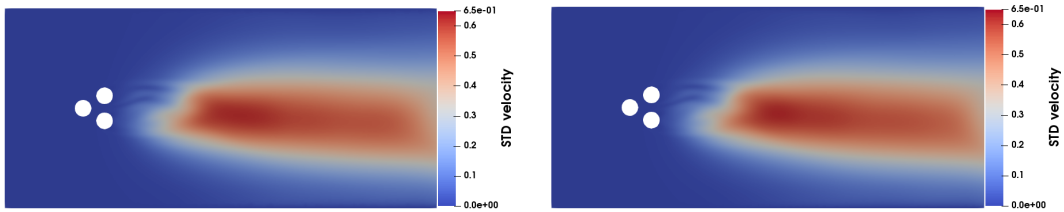


(c) Refined 1 mesh (43806 cells). **(d)** Refined 2 mesh (76552 cells).

Figure 10.2: Average velocity field of the mesh convergence simulations. The average velocity field is calculated on the last 100 time units with `.pvd` dumps for every 100 dt, corresponding to every 0.5 time units. During the 100 time units at $Re = 100$ we observe approximately 10 full vortex sheddings.



(a) Coarser mesh (16930 cells). **(b)** Simulation mesh (26480 cells).



(c) Refined 1 mesh (43806 cells). **(d)** Refined 2 mesh (76552 cells).

Figure 10.3: Standard deviation of the velocity for the mesh convergence simulations. The standard deviation is calculated on the last 100 time units in the same fashion as the mean velocity magnitudes presented in fig. 10.2.

From the results in table 10.2 and the plots in figs. 10.2 and 10.3 we can see that the results are very similar, independent of the mesh. In table 10.2 the coarse mesh differs by a relatively small margin, but compared to the difference between the 3 more refined meshes it is rather "large". The mesh

convergence simulation is run without any control on the cylinders, i.e. they are stationary. When we apply control the simulation might break if the mesh is not sufficiently refined, thus we run a simulation with active flow control with the 3 more refined meshes. As none of refined meshes break when flow control is applied we choose the mesh that has a fewest cells (less computational cost) and for which the mesh does not significantly influence the results. For simulations at $Re = 100$ the best combination is the mesh named "'Default" consisting of 26480 cells.

10.2.2 Mesh refinement at $Re = 150$

Simulations at higher Re makes the system more chaotic and is therefore an obvious route when we want to look for a more complex and possibly more difficult system for the DRL agent to control. Simulations at higher Re require more refined meshes, in addition to decreasing the numerical timestep, dt. The Reynolds number is increased by decreasing the value of the kinematic viscosity of the simulation.

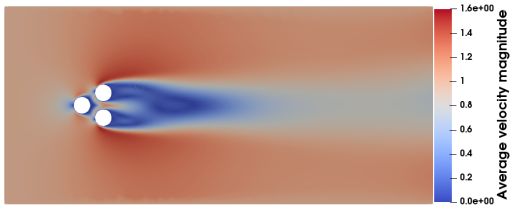
Following the same methodology as for $Re = 100$ we compare the three most refined meshes by running convergence simulations at $Re = 150$, with a refined numerical timestep $dt = 0.0025$.

Mesh name	$C_D \pm$ std	Deviation C_D	$C_L \pm$ std	Deviation C_L	# of cells
Default	3.5826 ± 0.0598	1.6692 %	0.0178 ± 0.1098	616.85 %	26480
Refined 1	3.5932 ± 0.0674	1.8758 %	0.0193 ± 0.1457	754.92 %	43806
Refined 2	3.5855 ± 0.0739	2.0611 %	0.0142 ± 0.1118	787.32 %	76552

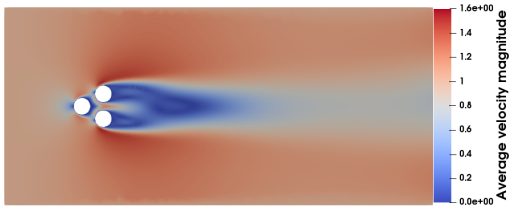
Table 10.3: Refinement levels of convergence study at $Re = 150$. Using numerical time steps of size $dt = 0.0025$. The mean and standard deviation values are calculated for the last 100 non-dimensional time units of the simulation. Deviation is calculated in the same way as for table 10.2. Compared to the convergence simulations at $Re = 100$ we can see a significant increase in the deviation of both coefficients. The flow at $Re = 150$ becomes more chaotic than the pseudo-periodic flow at $Re = 100$, causing larger variations in the drag and lift of the system.

As we can see the difference between the 3 meshes are larger than what we observed for $Re = 100$. In addition, the standard deviations are much larger for the higher Re case. The mesh we used for $Re = 100$ works fine when no flow control is applied, but the mesh is not sufficiently refined to handle flow control at the higher Reynolds number and the simulation crashes. The two more refined meshes are both able to handle active flow control. The most refined mesh, "Refined 2", is very computationally costly and thus we choose the mesh named "Refined 1" for the rest of our flow controlled simulations at $Re = 150$.

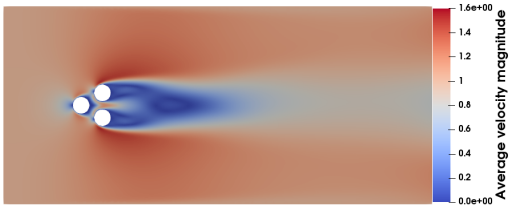
Figures 10.4 and 10.5 present the average velocity and standard deviation field of flow convergence at $Re = 150$. The fields are computed on the last 200 non-dimensional time units of the convergence simulations. The flow becomes more unstable and chaotic at higher Re, which is why we compute the average fields over a longer timespan than for $Re = 100$. This helps us avoid the average fields being significantly impacted by short instabilities that might appear at different times according to the mesh.



(a) $Re = 100$ simulation mesh (26480 cells).



(b) $Re = 150$ simulation mesh - Refined 1 (43806 cells).

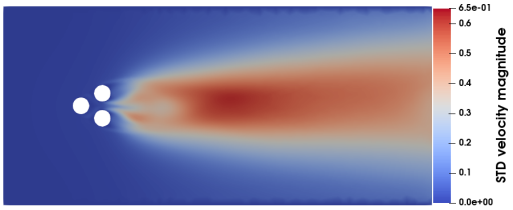


(c) Refined 2 (76552 cells).

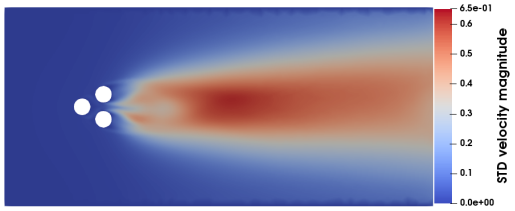
Figure 10.4: Average velocity field of the mesh convergence simulations. The average velocity field is calculated on the last 200 time units with `.pvd` dumps for every 200 numerical timestep, corresponding to every 0.5 time units. Due to the more chaotic flow at higher Re we choose to compute the average velocity and standard deviation for a longer interval than we did at $Re = 100$. As the flow is more unstable at higher Re computing the average velocity field for a shorter timespan could be significantly impacted by shortly lived instabilities in the flow.

10.3 Flow initialization

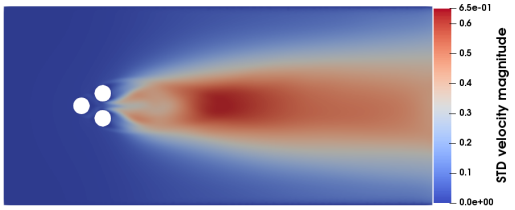
To perform active flow control with our DRL agent we need to properly initialize the flow field before we apply control. When we first create the mesh there is no flow, but we define an inflow of unit velocity $\mathbf{e}_x$ from the left. When starting the simulation from scratch the flow first takes on a transient flow structure as can be seen in fig. 10.6. For the flow to be properly initialized we have to let the simulation run for a long time, until the flow is stabilized. The stabilized flow regime in our case consists of a pseudo-periodic vortex



(a) Simulation mesh (26480 cells).



(b) Refined 1 mesh (43806 cells).



(c) Refined 2 mesh (76552 cells).

Figure 10.5: Standard deviation of the velocity for the mesh convergence simulations. The standard deviation is calculated on the last 200 time units in the same fashion as the mean velocity magnitudes presented in fig. 10.4.

shedding regime at $Re = 100$. At $Re = 150$ the flow is more chaotic, and although the periodic vortex shedding regime is still present, the drag coefficient does not follow the stable periodicity observed at $Re = 100$. Once the flow has been properly initialized we can load the saved flow configuration as an initial condition for the DRL simulations with AFC.

When the flow is fully developed we notice periodic vortex shedding happening behind the trailing cylinders. The vortex shedding alternates between developing behind each of the trailing cylinders as shown in fig. 10.7.

10.4 Active flow control setup

The system we will simulate to train PPO agents uses probes to sample either the pressure or the velocity of the flow to represent the flow as a *state* that is observed by the agent. The sampling probes are placed throughout the environment with a higher density of probes in the areas where the flow field varies the most, i.e. around the cylinders, and in the direct wake of the cylinders where we observe vortex shedding. The placement of the probes is given in fig. 10.8. In this project we have 476 probes sampling the flow, approximately three times the amount of probes used by Rabault et al. [33]. The number of probes could probably be reduced significantly without serious effect on the results. However, the agent is dependent on receiving a state that can sufficiently describe the flow, and reducing the number of probes by too much would lead to the agent having a harder time learning effective control

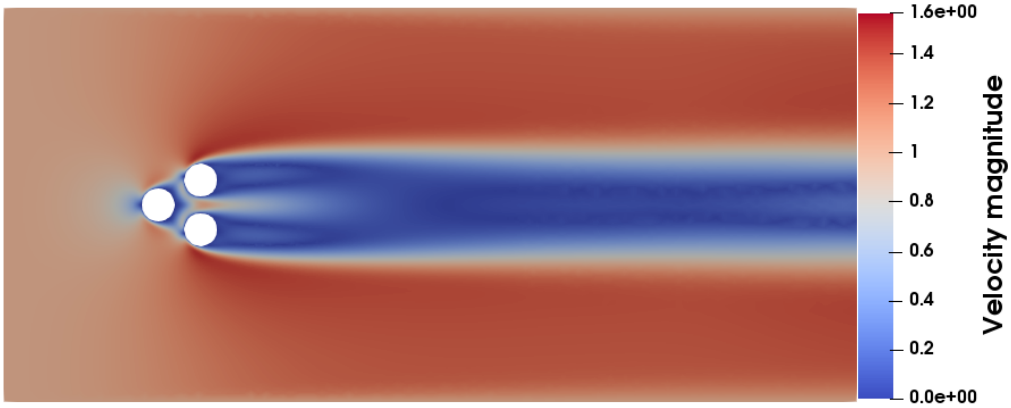
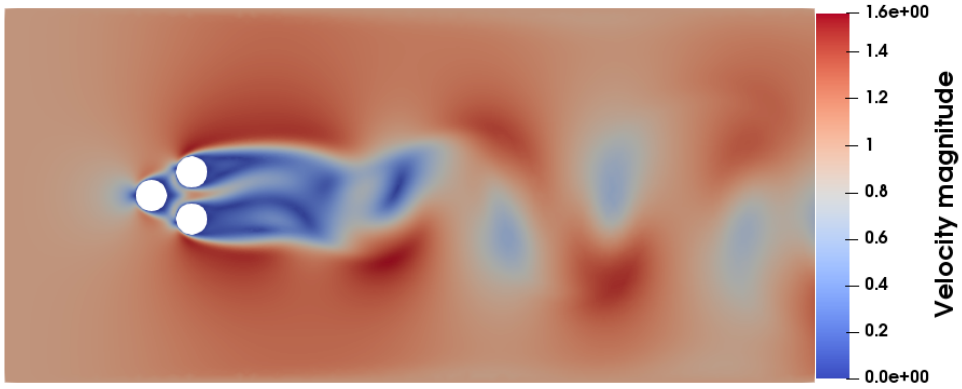


Figure 10.6: In the early stages of flow initialization the flow behaves very much like a laminar flow before developing into an unsteady pseudo-periodic regime with vortex shedding appearing behind the cylinders.

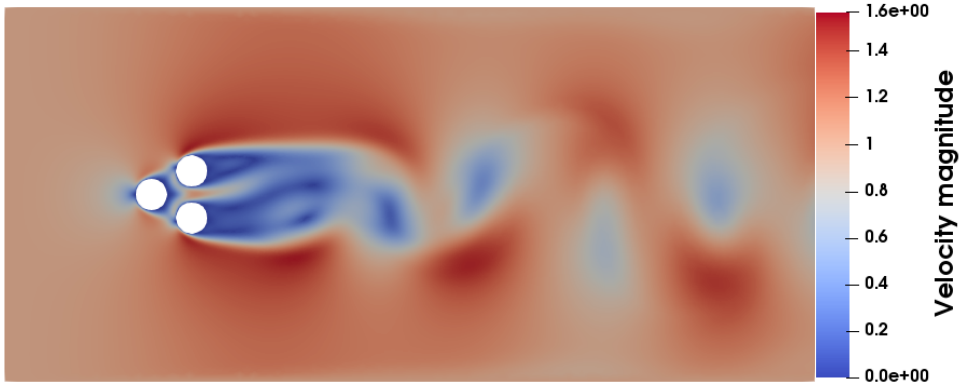
strategies.

We train PPO agents for a variety of configurations, most notably we apply control on simulations at two different Reynolds numbers ($Re = 200$ is experimental[1]). For each Re we train separate agents that seek to increase or reduce the drag of the system. For each configuration of a given Re with the goal of either reducing or increasing drag, we train two agents using two 80 or 160 actuations per episode of training. Having more actions per episode will reduce the time each actuation is applied, thus giving the agent the possibility to change the applied control more often. In section 6.1, Rabault et al. [33] found that letting the agent interact with the environment too often would lead to no learning because the agent would not be able to observe the effect of a single actuation input. However, if the flow changes in some way, e.g. the frequency of vortex shedding increases, more frequent and shorter lasting actuations can be beneficial, which we observe with simulations at $Re = 150$ where the flow is more chaotic. The difference in behavior between the baseline simulations at $Re = 100$ and 150 is illustrated in fig. 10.9.

[1]The mesh "Refined 2" is used in the experimental simulation at $Re = 200$. Due to the computational cost we did not perform rigorous mesh convergence for $Re = 200$.



(a) Vortex shedding appearing behind the top cylinder.



(b) Vortex shedding appearing behind the bottom cylinder.

Figure 10.7: Instantaneous flow field of the alternating vortex shedding developing behind the two trailing cylinders. Note that the vortex shedding is not entirely symmetric, but is more pronounced for the bottom cylinder. Because the flow is quite unstable we might observe that the flow changes which cylinder develops larger vortex shedding. Snapshots of $Re = 100$ simulation.

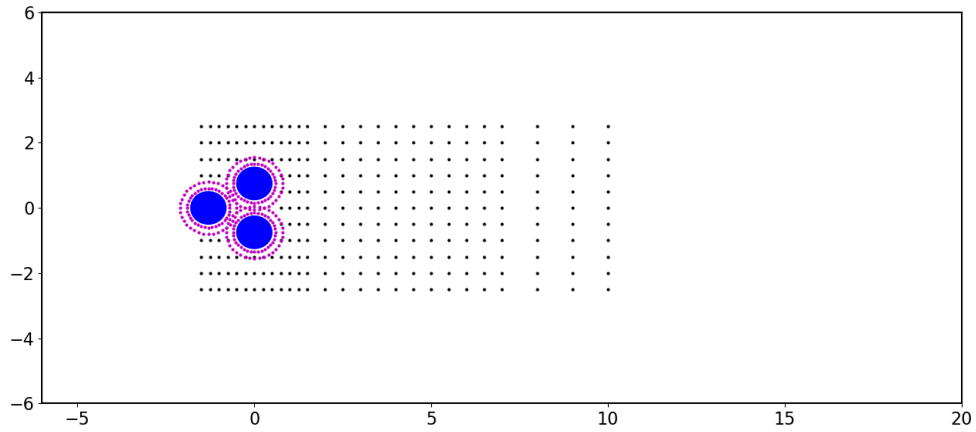
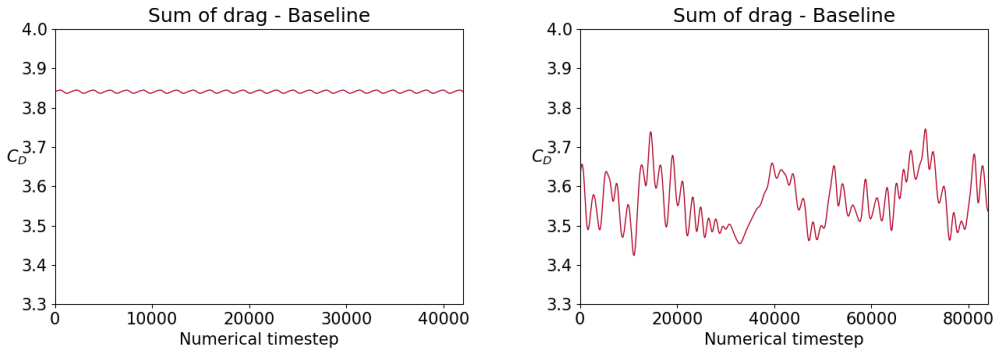


Figure 10.8: The pressure probes used to sample the flow field that represent the state observed by the DRL agent are placed frequently in the area around the cylinders, and more sparsely further away. Some probes are placed relatively far back from the cylinders because vortex shedding observed in these areas as well as close to the cylinder. The DRL agent is thus not given a complete state that represents the flow perfectly, but is rather given a sparser observation of the actual flow.

Re	# of actuations	duration per actuation	dt	# of cells in mesh
100	80	0.875	0.005	26480
100	160	0.4375	0.005	26480
150	80	0.875	0.0025	43806
150	160	0.4375	0.0025	43806
200	160	0.4375	0.0003125	76552

Table 10.4: Overview of the simulation parameters used in the application of DRL agents to control the flow of the fluidic pinball system. Each simulation configuration is used twice, first we apply control to reduce the drag and secondly we apply control to increase the drag. ($Re = 200$ is very computationally intensive and was only simulated once to increase drag).



(a) Baseline - $Re = 100$.

(b) Baseline - $Re = 150$.

Figure 10.9: Comparison of the sum of drag for a single run deterministic simulation without applied control at $Re = 100$ and $= 150$. The baseline flow at $Re = 150$ is not stable and behaves quite chaotically compared to the periodic behavior of the flow at $Re = 100$. Both simulations are run for the same non-dimensional duration. As the simulation at $Re = 150$ uses numerical timestep $\mathrm{dt}_{150} = \mathrm{dt}_{100}/2 = 0.0025$ the x-axis interval is $[0, 84000]$ vs. $[0, 42000]$.

Part III

Results

Chapter 11

Results - Fluidic Pinball

In this chapter we present the main results found by training a series of DRL/PPO agents to reduce and increase drag in the fluidic pinball system. We compare the control strategies of the PPO agents with simpler control strategies like constant actuations and sinusoidal control functions, which are chosen by approximating a constant value or sine functions to the strategy of the corresponding PPO agent. For each set of simulations we will present a selection of plots where we compare the different strategies and at the end of each subsection we present a table summarizing the average drag and lift coefficients of the simulations.

In section 11.1 we present our results of simulations at $Re = 100$. Starting with subsection 11.1.1 where we present the effectiveness of different methodologies at reducing the drag in the system. In subsection 11.1.2 we present our findings for increasing drag at $Re = 100$. Section 11.2 includes results of simulations at $Re = 150$, where the flow is more chaotic than what is observed for $Re = 100$. As in the previous section, we start by presenting drag reduction results in subsection 11.2.1, and then the results of drag increase strategies in subsection 11.2.2. The final section of the chapter contains power spectral density (PSD) plots with a discussion of how applying flow control may change the governing frequencies determining the vortex shedding oscillations.

11.1 Active flow control $Re = 100$

Once we have initialized our flow as described in section 10.3 we can start training the DRL agent to control the flow. We start looking at the fluidic pinball system at $Re = 100$. By modifying the reward function applied to the DRL agent we guide the agent to either decrease or increase the drag of the system. The agent is able to find strategies to both problems which will be presented in the following sections. At $Re = 100$ we train the agent

for 800 episodes where an episode lasts for 70 non-dimensional time units, corresponding to approximately 8 vortex sheddings. For each episode we allow the DRL agent to update the control rotations of the cylinders, 80 or 160 times. These rotation updates are carried out similarly to what is described by fig. 6.3, where we carry out several numerical steps with the same applied rotation before the agent is allowed to update the rotations.

Initial simulations were done using 80 actuations, as an input of 10 actuations per vortex shedding has proven effective in previous projects. We also carry out simulations using 160 actuations per episode, i.e. 20 actuations per vortex shedding, which might be a benefit in certain flow systems.

11.1.1 Drag reduction

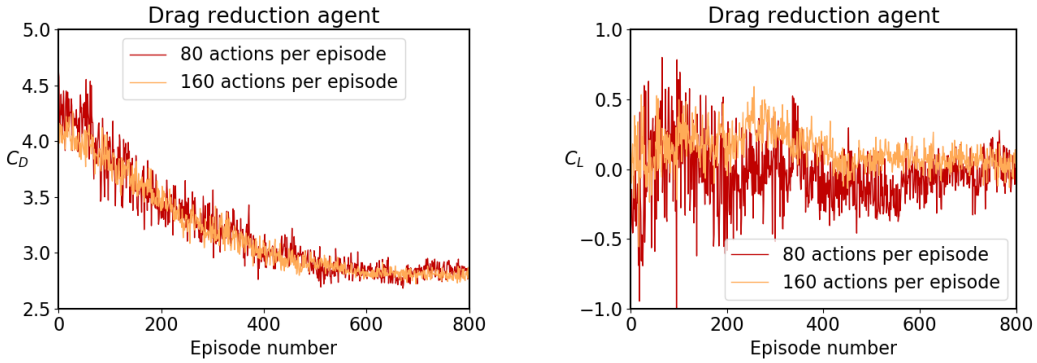
The first flow control agent is trained to reduce the drag observed on the cylinders. We introduce a reward function that penalizes drag values while also discouraging high lift values. In Rabault et al. [33] it was observed that no lift penalization lead to very good drag reduction, but also caused a significant increase in lift that would be damaging in most practical applications. The reward function implemented in the DRL code is given as

$$r_t = -\langle C_D \rangle - \alpha |\langle C_L \rangle|, \tag{11.1}$$

where $\langle \cdot \rangle$ is a moving average over the current given action and α is weight parameter penalizing increased lift in the system. For our simulations we used a value of $\alpha = 0.2$. $\alpha = 0.5$ was also tested, but the larger lift penalty discouraged nearly all cylinder rotation, i.e. no actual applied control.

Once the reward function has been determined we start training the DRL agent, letting the agent interact with the simulation and discover strategies for reducing the drag. During training we observe the accumulated drag and save the average drag for each episode. If the average drag for each episode is randomly fluctuating it is clear that the agent is not able to find a strategy for control. However, if the average drag per episode gradually converges to a some value lower than the initial average drag it is clear that the agent has found a strategy for drag reduction. In fig. 11.1 we plot the average drag and lift per episode.

Once the agent has obtained robust and stable learning we run a simulation with deterministic prediction to obtain results from the agent without the exploration noise that is present during training. Instead of repeating each time a simulation is done that "deterministic prediction" was used, we will call such simulations a `single runner` simulation, or a `single run` simulation. The naming convention, `single run` is simply derived from the python script for running a simulation with deterministic prediction which is named `single_runner.py`. During training we repeatedly do shorter simulations



(a) Average drag coefficient per episode vs. episode number.

(b) Average lift coefficient per episode vs. episode number.

Figure 11.1: Figure 11.1a shows robust learning taking place where the average drag coefficient over an episode steadily decreases until around episode $5 - 600$ and is stable for the rest of training. The figure contains results for both agents, updating the applied control 80 or 160 times per episode. Figure 11.1b presents the average lift coefficient per episode which can give an indication of whether the average flow is symmetric while control is applied or if the applied control will cause asymmetry to appear.

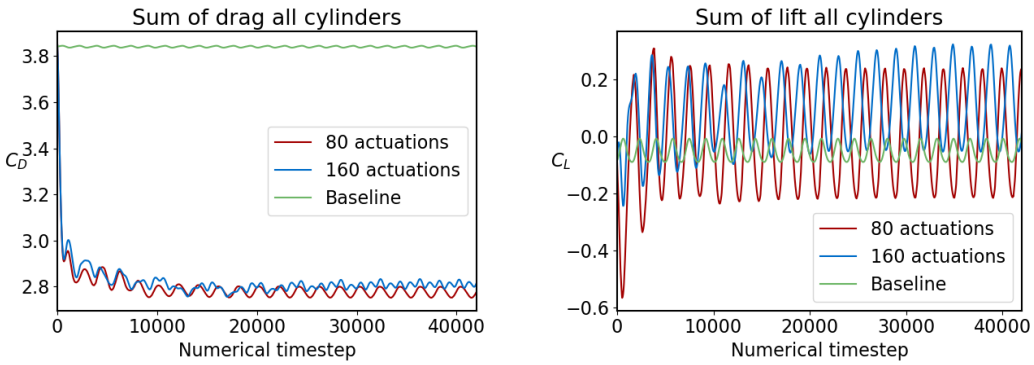
until the agent is trained, while for a single run simulation we run one single longer simulation equal to 3 times the length of a training episode, i.e. $70 \times 3 = 210$ time units, or $14000 \times 3 = 42000$ numerical time steps.

In fig. 11.2a we see that both agents are able to significantly reduce the drag in the system. Figure 11.2b shows us that the lift of the baseline simulation is periodic and $C_L \approx 0$, while when we apply control to reduce drag this comes at the cost of an increased amplitude for the periodic oscillations of the lift.

As the DRL agents are able to reduce the drag we want to investigate the control strategies they apply. Figure 11.3 shows that both agents find very similar strategies for achieving drag reduction. The first few thousand numerical timesteps consist of non-periodic actuations which don't seem to follow any obvious strategy. However, after the initial actuations the strategy develops to a pseudo-periodic strategy for each cylinder. The first cylinder, cylinder 0, is given periodic inputs of magnitude close to 0, while the two trailing cylinders apply actuations of opposite direction, with similar magnitudes much larger than the cylinder in front.

In addition to comparing the two different DRL agents in fig. 11.2 with the baseline flow we also compare the agents with corresponding simulations using:

1. Constant control equal to average actuations of DRL agents. (Mainly



(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

Figure 11.2: We compare the results of single run deterministic simulations at $Re = 100$. A baseline flow without applied flow control is plotted with two DRL agents seeking to reduce the drag of the system. Both DRL agents use the reward function given by eq. (11.1). The agents are exactly equal except for the number of actuations per episode which are set to 80 and 160, respectively. Note that when using 160 actuations per episode each actuation is applied for half the duration of an actuation from the 80 actuations per episode agent.

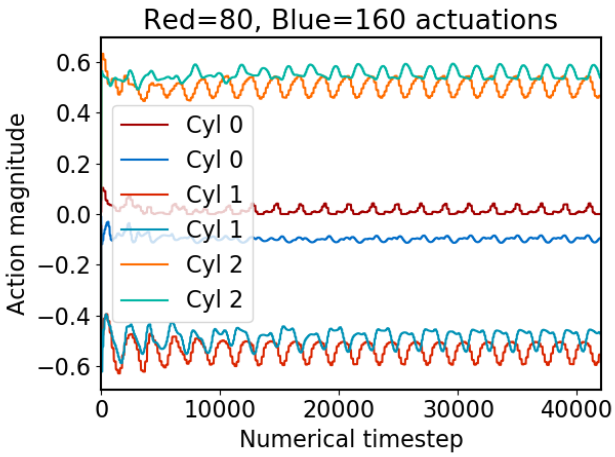
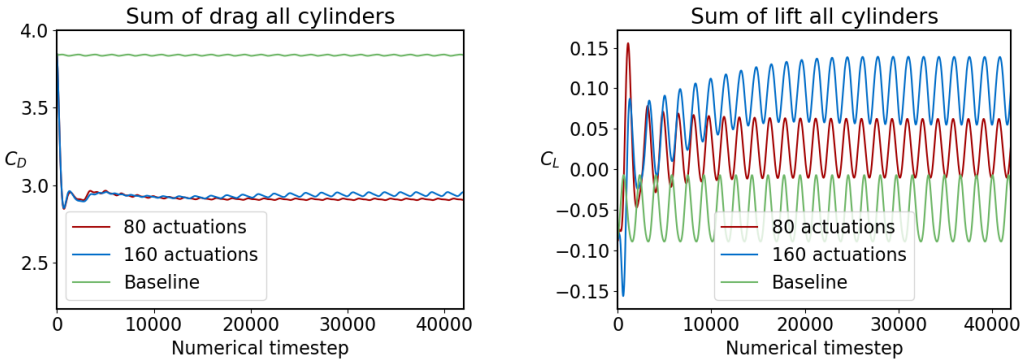


Figure 11.3: The actions taken by the two DRL agents in fig. 11.2 are plotted together for direct comparison. The y-axis measures the magnitude and the direction of a cylinders rotation, while the x-axis is again the numerical timestep of the simulation. The lines in red shades are the controls applied by the agent using 80 actuations, while the blue lines are for the agent using 160 actuations per episode, i.e. per 14000 numerical timestep.

applied for drag reduction.)

2. Sinusoidal control strategy fitted to DRL agent actuations. (Applied for both drag reduction and increase.)



(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

Figure 11.4: Drag and lift coefficients for simulation using constant rotations to control the flow. The constant rotations are calculated by taking the average magnitude of the control actuations applied by the two DRL agents in fig. 11.3. Each cylinder is then given the corresponding average control actuations as a constant control actuation, i.e. the same rotation is applied throughout the entire single run simulation. From fig. 11.4a it is obvious that constant rotations are able to significantly reduce the drag coefficient. Figure 11.4b indicates that the constant rotations cause a small change in the total lift coefficient, but compared to the DRL agents, the magnitudes of oscillations are relatively small.

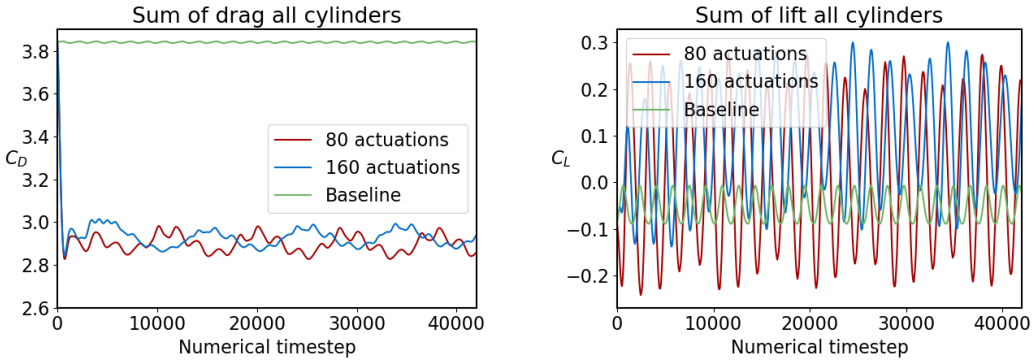
Significant drag reduction compared to the baseline simulation is observed in fig. 11.4a. The DRL agents perform slightly better, but constant actuations prove to be an effective alternative control method. However, in order for constant control to be a real option we need a way to find what the actuations should be independent of DRL agents. For some systems it might be possible to find effective constant actuation values through a grid- or random search method, but for more complex systems this becomes more and more costly. A researcher or engineer with considerable skill and intuition of fluidic mechanics might be able to find useful control values for simpler systems, but this will once more become more difficult as the complexity of the system increases.

Sinusoidal control actuations can be considered as a step up in complexity compared to constant actuations. The rotation of the cylinders is not constant any longer, but varies according to an individually determined sine function dependent on the actuation number. By using a discrete fourier transform from `numpy` combined with a least squares curve fitting tool from `sympy` we find sine functions approximating the control strategy of the DRL agents.

The sinusoidal control actuations are updated with the same frequency as the DRL agent they are fitted to. Each cylinder is given a sine function of the form:

$$A \sin(\omega \cdot x + \varphi) + C, \tag{11.2}$$

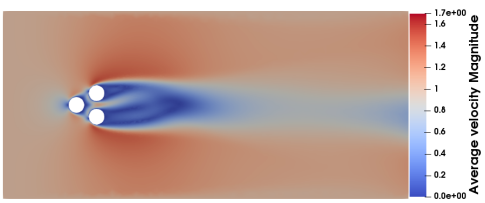
where A is the amplitude, ω is the angular frequency, x is the actuation number, φ is the phase, and C is a constant offset that determines the value the sine wave oscillates around. The results of reducing drag by using sine functions to control the rotation of the cylinders are presented in fig. 11.5.



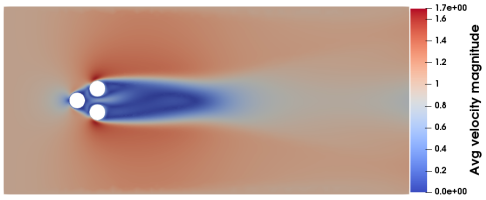
(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

Figure 11.5: Drag and lift coefficients for simulations using sinusoidal functions to control the rotation of each cylinder. Sinusoidal control functions perform slightly better than using constant actuations and slightly worse than controlling the flow with DRL agents, but with significantly more variations in the drag coefficient than observed for the other methods. We also observe an increased amplitude in lift coefficient oscillations, clearly illustrated in fig. 11.5b

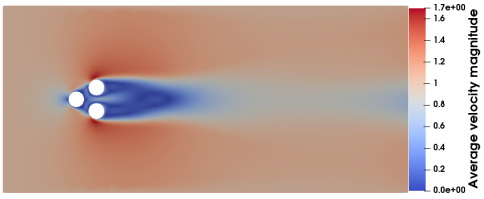
From figs. 11.2a, 11.4a and 11.5a it is clear that reducing drag in the fluidic pinball system is very much possible, but to understand what is happening we will also take a look at the velocity profile of the flow, after control is applied. During single run simulations we save the instantaneous flow to `.vtu/.pvd` files which are accessible with ParaView once the simulation has ended. For $Re = 100$ we saved the flow once every 50 timesteps corresponding to every 0.25 time unit. Using the ParaView software we can compute the mean and standard deviation of the flow. Below we present a side-by-side comparison of the computed mean flow with corresponding standard deviation for one DRL agent, one flow controlled by constant actuations, one sinusoidally controlled flow, and the baseline flow.



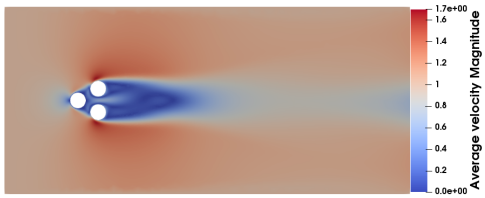
(a) Baseline mean flow.



(b) DRL agent 80 actuations.



(c) Constant 80 actuations.



(d) Sinusoidal 80 actuations.

Figure 11.6: We compare the mean flow of the baseline simulation with three different drag reduction control methods. Figure 11.6b shows a clear increase in the size of the recirculation area (the area of low velocity behind the cylinder) compared to the baseline, which is associated with lower pressure drop behind the cylinder. The recirculation area size for constant and sinusoidal control is not increased in the same way as for DRL control, but are somewhat more symmetric than for the baseline simulation.

Control strategy	$C_D \pm$ std	$C_L \pm$ std	C_D reduction
Baseline	3.8407 ± 0.0029	-0.0523 ± 0.0294	0 %
DRL 80 actions	$\mathbf{2.7764 \pm 0.0170}$	$\mathbf{0.0011 \pm 0.1607}$	**27.71** %
DRL 160 actions	2.8056 ± 0.0125	0.1136 ± 0.1271	26.95 %
Sinusoidal actions - 80	2.8943 ± 0.0379	0.0238 ± 0.1541	24.64 %
Sinusoidal actions - 160	2.9274 ± 0.0334	0.1009 ± 0.1143	23.78 %
Constant actions - 80	2.9078 ± 0.0031	0.0251 ± 0.0258	24.29 %
Constant actions - 160	2.9390 ± 0.0091	0.1002 ± 0.0295	23.48 %

Table 11.1: The final drag and lift coefficients of the different control strategies created to reduce drag at $Re = 100$ are presented. Each value is calculated as the mean value over the last half of a single run evaluation simulation. The control strategy we determine gave the best results is highlighted in bold. The drag reduction in % is calculated as $(C_{D,\text{baseline}} - C_{D,\text{control}})/C_{D,\text{baseline}}$.

11.1.2 Drag Increase

After the successful application of DRL agents to reduce drag we want to see if the agents are able to work the other way, i.e. increase the drag in the system. We need to implement a new reward function that will reward increased drag and penalize low drag values. Similar to the drag reduction reward function we again add a small term penalizing increased lift. After testing a few different reward function options the best one also includes a penalization term which is supposed to discourage large actuation magnitudes. The reward function, dubbed `more_drag_ simple_actuation` in the code, can be written as:

$$r_t = \langle C_D \rangle - \alpha |\langle C_L \rangle| - \beta \sqrt{\sum_i Q_i^2}, \tag{11.3}$$

where $\alpha = 0.2$, same as for drag reduction, $\beta = 0.1$, and $\langle \cdot \rangle$ denotes the moving average over the last given action.

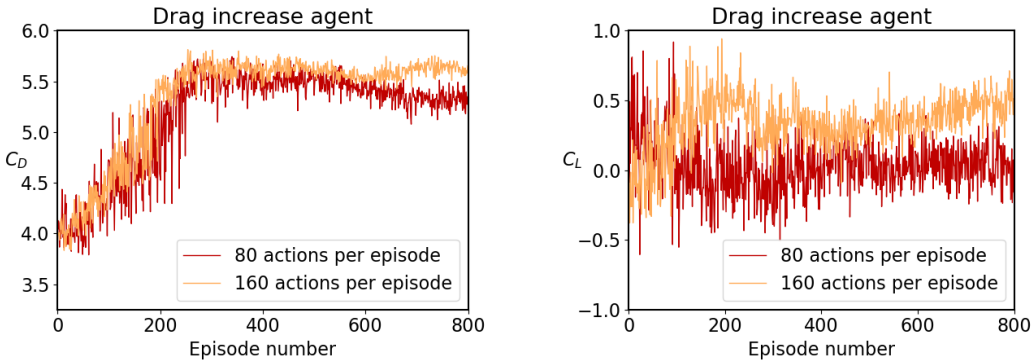
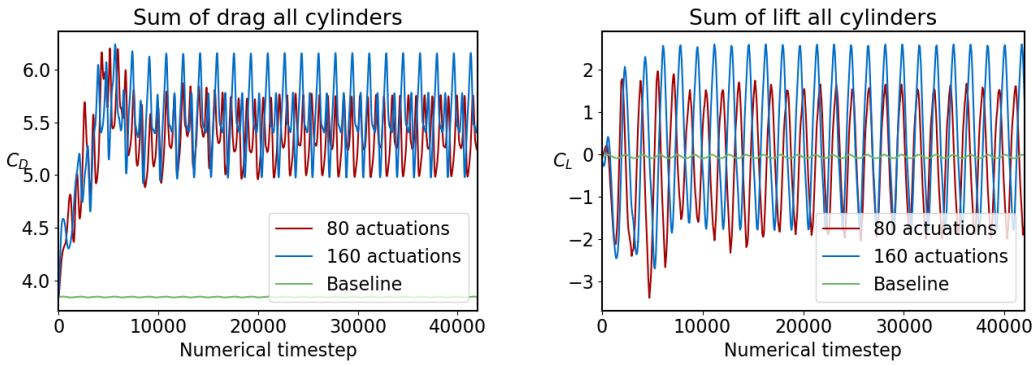


Figure 11.7: Drag and lift coefficient average per episode while training the DRL agent to increase the drag in the system. Comparison between using 80 and 160 control actuations per episode.

After approximately $350 - 400$ episodes the agents are converged and finished learning. From fig. 11.7 we can see that there are smaller variations from episode $6 - 800$ where the agent applying 160 actuations per episode seems marginally better than the 80 actuations agent. Note that as for drag reduction the training curve includes noise due to exploration which and that to evaluate the final model we need to run a simulation using deterministic prediction. The results of those simulations, named *single run* simulations, are presented in fig. 11.8.

Figure 11.8a shows us clearly that the DRL agents are very capable of increasing drag in the fluidic pinball when we change the reward function. Both agents quickly increase the drag in the first 10000 numerical timesteps



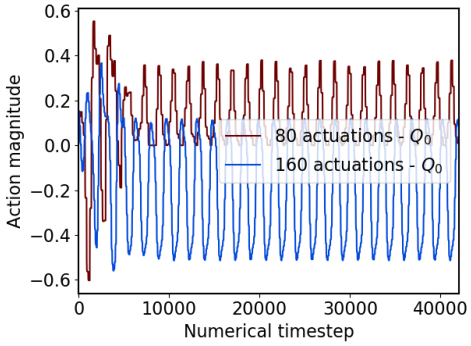
(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

Figure 11.8: Comparison of two DRL agents. The drag and lift coefficient of the baseline flow without applied flow control is plotted with the results of two DRL agents seeking to increase the drag of the system. Both DRL agents use the reward function given by eq. (11.3). The configuration is the same except for the number of actuations per episode which are set to 80 and 160, respectively and which also changes the duration of each actuation.

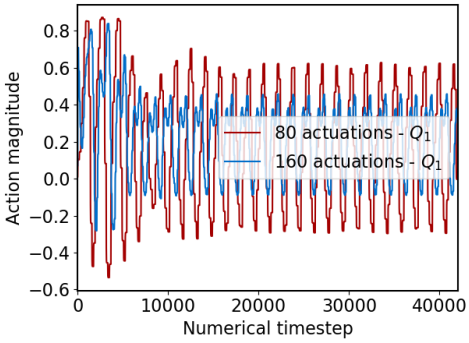
after which the drag fluctuates heavily. The agent applying 160 actuations per episode achieves slightly more drag, with similar drag coefficient minimums, but with higher maximums. From fig. 11.8b we can clearly see that the control strategy is affecting the lift in the system quite drastically. Both agents cause large periodic increases in lift indicating that obtaining increased drag is not easily possible without introducing increased lift. To understand how the agent is able to increase the drag we need to look at the control actuations, i.e. the rotations of the cylinders. In fig. 11.9 we compare the control actuations taken by the two agents, one plot per cylinder.

From fig. 11.9 it is not trivial to understand exactly what the control actuations actually do to the flow. This is a common issue within deep learning because of the difficulties researchers meet when trying to understand exactly what a NN has learned. In our case we have the advantage of being able to see the physical system the DRL agent interacts with, and as such can observe how the flow develops when control is applied. In fig. 11.10 we can see that the control actuations causes significantly stronger and more frequent vortex shedding, compared to the baseline vortex shedding in fig. 10.7.

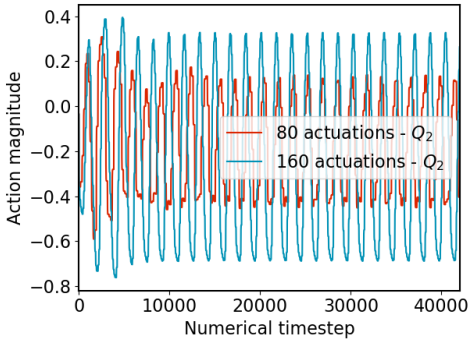
The control strategy causing oscillating vortex shedding was initially suspected to come as a results of the lift penalty in the reward function which penalizes the agent if the mean lift drifts from zero, but this is not the case. We simulated the fluidic pinball system applying constant actuations in the same direction for both the trailing cylinders, using a magnitude of 0.8, equal to 80% of the maximum rotation magnitude and observed a drag increase



(a) Rotations applied to the first cylinder.



(b) Rotations applied to the bottom trailing cylinder.

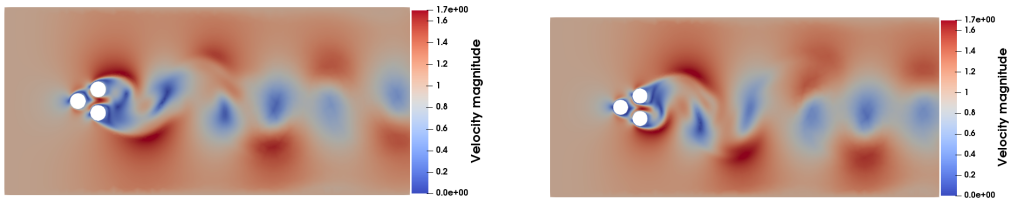


(c) Rotations applied to the top trailing cylinder.

Figure 11.9: Actions taken by DRL agents to increase drag. Both agents use actuations of larger magnitude than seen in fig. 11.3, and in contrast to the drag reduction strategies the first cylinder is also given rotations of significant magnitude.

with periodic oscillations of the drag coefficient of ≈ 4.2. This means that controlling the flow and obtaining increased drag using constant actuations is possible, but requires that both trailing cylinders rotate in the same direction. This strategy also causes a massive increase in lift with an approximate average lift coefficient of $\approx (-)4.2$. (The negative sign indicates direction and is dependent of the direction of rotation we choose). Comparing the resulting drag coefficient $C_D \approx 4.2$ to the results of the DRL agents in fig. 11.8a it is clear that a constant actuation strategy to increase drag is significantly less effective than a control strategy determined by a DRL agent.

In subsection 11.1.1 we compared the baseline flow with controlled flows where we apply control by DRL agents, constant actuations, and sinusoidal control functions. In the previous paragraph we determined that constant actuations are is not an effective strategy to increase the drag, and will thus focus our comparison on DRL agents and sinusoidal control versus the base-

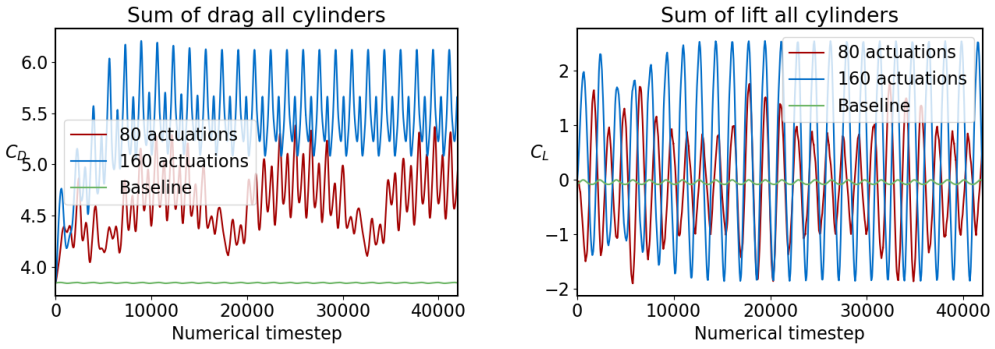


(a) Strong rotations clockwise cause vortex shedding from the top trailing cylinder.

(b) Strong rotations anti-clockwise cause vortex shedding from the bottom trailing cylinder.

Figure 11.10: The two snapshots of the instantaneous velocity magnitude show the controlled flow at two extremes when the flow is controlled by a DRL agent to increase drag. The periodic rotations applied by the DRL agent cause strong vortex shedding behind the two trailing cylinders in a periodic fashion according to the periodic actuations.

line flow.

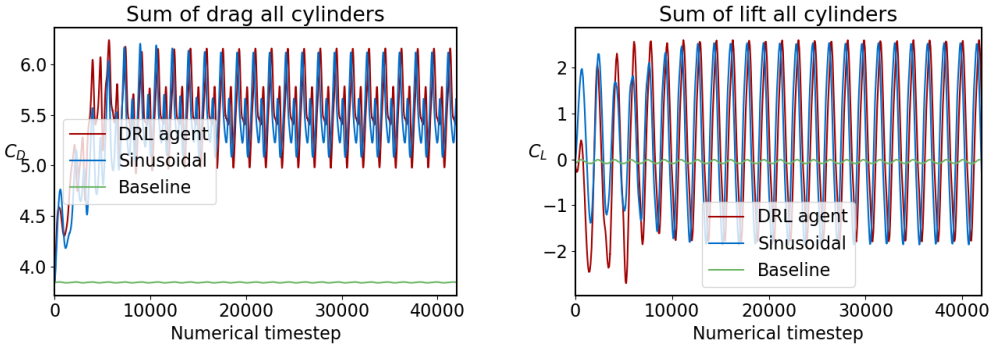


(a) Drag coefficient vs. numerical timestep.

(b) Lift coefficient vs. numerical timestep.

Figure 11.11: Flow controlled by sinusoidal functions compared to baseline flow. The sinusoidal functions fitted to the two DRL agents are both able to increase the drag, but interestingly the sinusoidal functions fitted to the 80 actuations per episode DRL agent is not stable in the same way as for the corresponding 160 actuation sinusoidal functions. However, the lift is increased quite significantly more in the case of the 160 actuation sinusoidal function, which follows closely what was observed for the DRL agents in fig. 11.8b. In fig. 11.8a the DRL agent applying 160 actuations per episode performed better than the 80 actuations per episode agent, and again we observe that the control method with more frequent actuation updates perform better.

From fig. 11.11a it is clear that a sinusoidal control strategy can be very effective. As the results of the best DRL agent and the best sinusoidal control strategy at a glance are very similar we plot them together for a direct comparison.



(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

Figure 11.12: Comparison of DRL agent, sinusoidal control and baseline. The difference between the two control strategies are very small, but in addition to a small phase shift between the two graphs the DRL agent also has marginally larger amplitude in its oscillations. One thing to note is that the DRL agent seems to reach higher drag coefficient values in a shorter amount of time, but after approximately 10000 timesteps the two control strategies are almost identical. The fact that both methods reach the same drag coefficient values indicates that the non-periodic initial actuations of the DRL agent are not needed to reach the equal levels of drag increase, but is rather used to speed up "convergence" to the periodic flow of maximum drag (according to the DRL agent).

In conclusion the DRL agent is very capable when trying to increase the drag in the system, but at the same time individual sinusoidal control functions are just as effective at increasing the drag. It is however important to note that the effective sinusoidal control functions are fitted from the strategy the DRL found during training, and that finding the corresponding control functions independently would very difficult. We also discovered that the most effective strategy is to create oscillating vortex shedding from each of the trailing cylinders, and that constant rotations of high magnitude in the same direction will increase the drag, but not by as much, in addition to creating a lot of lift in a single direction.

Control strategy	$C_D \pm$ std	$C_L \pm$ std	C_D increase
Baseline	3.8407 ± 0.0029	-0.0523 ± 0.0294	0 %
DRL 80 actions	5.3685 ± 0.2344	-0.1025 ± 1.1920	39.78 %
DRL 160 actions	$\mathbf{5.5629 \pm 0.2992}$	$\mathbf{0.4505 \pm 1.4529}$	**44.84** %
Sinusoidal actions - 80	4.7011 ± 0.2820	-0.0691 ± 0.8137	22.40 %
Sinusoidal actions - 160	5.4872 ± 0.2895	0.4376 ± 1.5056	42.87 %
Constant actions - same direction	4.1869 ± 0.1164	-4.1808 ± 0.4421	9.01 %

Table 11.2: The final drag and lift coefficients of the different control strategies created to increase drag at $Re = 100$ are presented. Each value is calculated as the mean value over the last half of a single run evaluation simulation. The control strategy we determine gave the best results is highlighted in bold. The drag increase in % is calculated as $(C_{D,control} - C_{D,baseline})/C_{D,baseline}$.

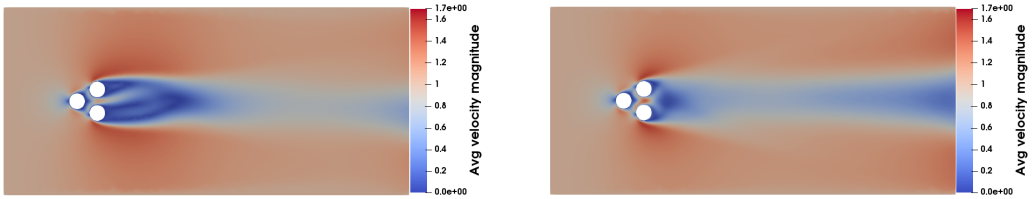
11.2 Active flow control $Re = 150$

After successfully finding strategies to increase and decrease drag using DRL agents at $Re = 100$ we want to look at a more complex system. We could look at a more complex system with e.g. more cylinders, but simulating for other Reynolds number (Re) values will result in a more chaotic and complex flow, and will make comparing the control strategies of the different agents easier. For complex problems training a DRL agent can be challenging, and in such cases using a pre-trained model, as introduced in section 3.6, can be very helpful. When controlling the flow at $Re = 150$ the agent had trouble converging without any pre-training. By training a DRL agent at $Re = 100$ for a shorter time than what is needed to reach full convergence we give the agent some starting help, while at the same time not training it for so long that it is locked into the exact same strategy as for a fully converged model at $Re = 100$. In our case we trained the agent for 150 episodes at $Re = 100$ before then training the agent for 650 episodes at $Re = 150$.

When moving to a higher Reynolds number we need to initialize the flow once more with the new simulation parameters. Note that the flow starts from a state of lower drag than for Re. For $Re = 150$ we have $C_D \approx 3.6$ while for $Re = 100$ the drag coefficient converges to $C_D \approx 3.8$ during initialization. Once we have established that the flow is not evolving any further we save the current state after 1200 time units and load that state as the starting flow when applying control.

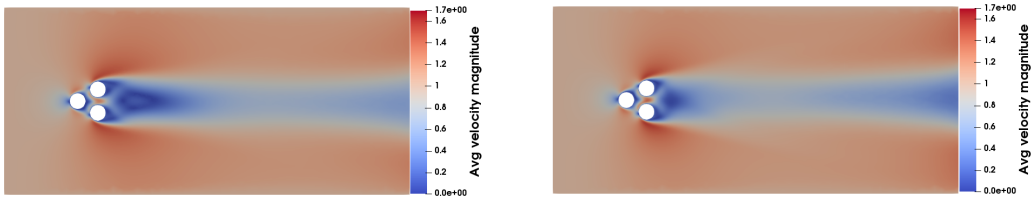
11.2.1 Drag reduction

As in section 11.1 we first want to reduce the drag in the system. The first step in the process is to pre-train the agents at $Re = 100$. We train two agents



(a) Baseline mean flow.

(b) DRL agent 160 actuations.



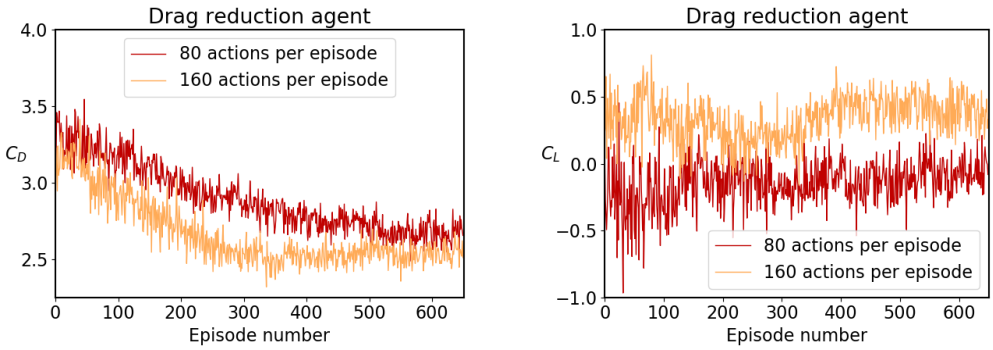
(c) Sinusoidal 80 actuations.

(d) Sinusoidal 160 actuations.

Figure 11.13: We compare the mean flow of the baseline simulation with three different drag increase control methods. Significant differences are obvious if we compare the mean flow for drag increasing strategies with the mean flows of drag reduction methods in fig. 11.6. The recirculation area, i.e. the area behind the cylinders with very low velocity (color-coded blue), is growing during drag reduction, while during drag increase this is reduced. Figures 11.13c and 11.13d illustrates this, where the first figure corresponds to a somewhat failed strategy with mean $C_D \approx 4.70$, compared to the second best strategy of all at drag increase at $Re = 100$, reaching mean $C_D \approx 5.49$.

at $Re = 100$ for 150 episodes, using the same reward function we used in the previous section, eq. (11.1). Again we let one agent apply 160 actuations per episode and the other applying 80 episodes. The idea of using more actuations was found when the first simulations at $Re = 150$ were being carried out as they continuously struggled to converge. By investigating the flow initialization simulation it was discovered that the higher Re flow experiences more frequent vortex shedding, reducing the number of actuations per vortex shedding. The agent will thus have less actuations to counter the vortex shedding, and will rely on taking smarter actuations which are more effective throughout the longer duration each actuation lasts. In fig. 11.14 we compare the mean drag per episode of two DRL agents using 80 and 160 actuations per episode.

Figure 11.14 tells us that with the help of pre-training the DRL agents are

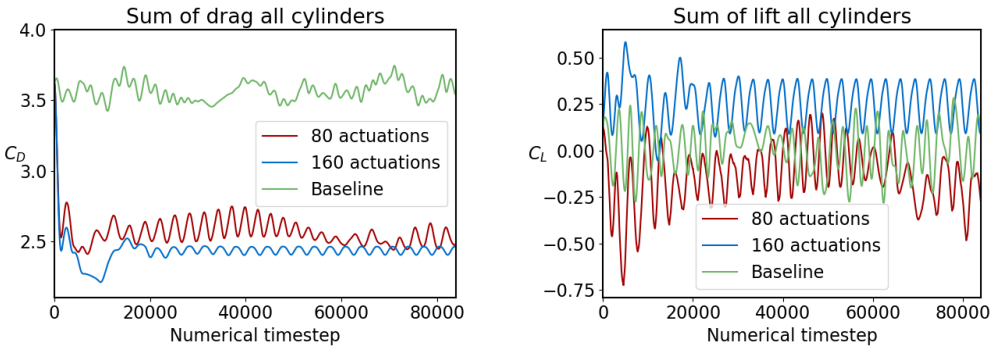


(a) Drag coefficient vs. Episode number **(b)** Lift coefficient vs. Episode number

Figure 11.14: Average drag and lift coefficient per episode while training the DRL agent to reduce drag in the system. The pre-trained agent using 160 actuations per episode converges a lot faster than the 80 actuations agent, but with quite similar results judging from this figure. Remember that the final judgement should be reserved until we have studied the single run (deterministic prediction) simulation, which does not contain any exploration noise as the results in this figure does. Note that both agents have been pre-trained for 150 episodes at $Re = 100$. Interestingly the average lift coefficient is quite different between the two agents, following a similar pattern as for $Re = 100$ where the agent of 160 actuations during training moves further away from $C_L = 0$.

able to learn how to control the flow at $Re = 150$ in order to reduce drag. Once learning is finished we need to evaluate the control strategy the agents have come up with by running a new simulation with deterministic predictions (single run), meaning no exploration noise is applied to the actuations determined by the agent. Because of the more complex flow and refined mesh we need to reduce the numerical timestep from $dt = 0.005$ to $dt = 0.0025$. The simulation length is kept the same, but due to the refined numerical timestep the new plots will have an x-axis of numerical timesteps $[0, 84000]$ compared to $[0, 42000]$ for $Re = 100$. We start by comparing the single run results of the two DRL agents before moving on to comparing the DRL results with simpler strategies like constant rotations and sinusoidal control functions.

From fig. 11.15a we observe clear drag reduction using DRL agents at $Re = 150$. It is also apparent that the differences between the agents are more significant at $Re = 150$ than what we observed at $Re = 100$. Most notably the drag coefficient of the flow controlled by the 80 actuations agent does not reach a stable periodic flow state, but instead oscillates somewhat arbitrarily, although at a much lower value than the baseline. The agent updating rotations 160 times per episode is able to stabilize the flow in addition to reducing the drag coefficient more than the other agent. Figure 11.15b reinforces this,



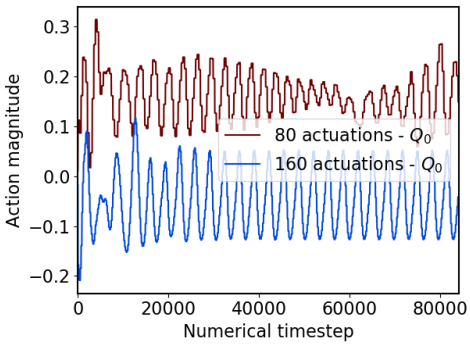
(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

Figure 11.15: Drag reduction flow control by DRL agents compared to baseline flow. The baseline at $Re = 150$ is not a stable periodic flow with small oscillations as observed for $Re = 100$, but is instead quite unstable and non-periodic. The DRL agent with 160 actuations per episode performs better than the 80 actuations per episode agent, and is able to keep the flow in a stable periodic state. The 80 actuations agent also reduces drag quite nicely, but is not able to obtain the same stable oscillating behavior as the 160 actuations agent. For the lift coefficient we see that for the baseline flow and the flow controlled by the 80 actuations agent oscillate quite wildly. On the other hand, the lift coefficient of the flow controlled by the 160 actuations agent oscillates nicely around a value $C_L \approx 0.25$ after ≈ 30000 numerical timesteps.

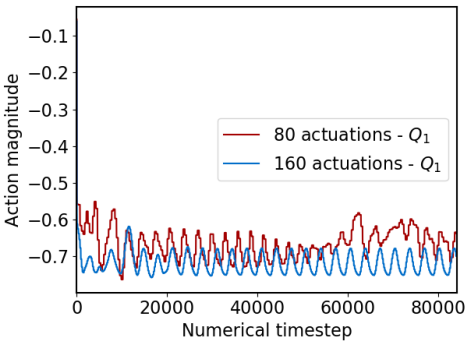
where the lift coefficient is stabilized and reaches a stable oscillating state, compared to the relative chaotic oscillations observed for the baseline and the 80 actuations agent. In fig. 11.16 we compare the rotations of the two agents for each cylinder, which might give us an indication to what causes the differences of the two agents.

In fig. 11.16 we compare the rotations applied to the three cylinders by the two DRL agents trained to reduce drag at $Re = 150$. For the first agent, updating the rotations 80 times per episode, we can see that the irregular rotations coincide with the interval where the drag coefficient in fig. 11.15a is closest to the strategy of the 160 actuations agent. However, after a short while of very low drag coefficient values the drag increases once more. The agent might seem to be "greedy" where it discover a set of rotations that over a short span of time will give very low drag values, but that the agent is not able to stabilize the flow in such a regime. On the other hand the agent using 160 actuations is able to stabilize the flow in a very efficient regime where stable oscillating rotations with small amplitudes keep the drag stable and low.

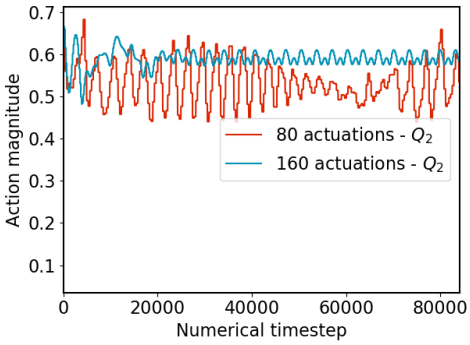
After having found the better strategy of the DRL agents we look at the



(a) Rotations applied to the first cylinder.



(b) Rotations applied to the top trailing cylinder.

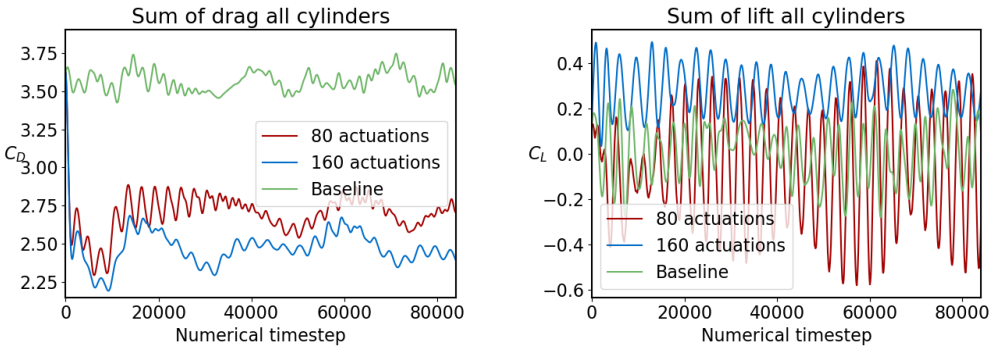


(c) Rotations applied to the bottom trailing cylinder.

Figure 11.16: Comparing the control strategies of the two DRL agents at $Re = 150$. The control strategy of the 80 actuations DRL agent develops in a very strange fashion where the rotations start off as stable oscillations, but towards the later stages of the simulation becomes very irregular. This coincides with the time of simulation where the stable oscillating behavior of the drag coefficient breaks. The second agent of 160 actuation updates per episode applies more stable rotations, that oscillate in the same way from very early on in the simulation, obviously having found the optimal rotations needed to keep the flow stable and reducing the drag coefficient.

two other methods of flow control applied in this project, namely constant actuations and sinusoidal control functions. The sinusoidal control functions in fig. 11.17 are fitted to the control strategy of the DRL agents presented in **??**. Due to the irregular rotations of the 80 actuations agent it was not possible to find a sensible fit on all rotations in the interval $[21000, 84000]$. However, by fitting a sine function to the interval where the actuations are similar to a sine function, $[21000, 42000]$, we are able to get a good fit.

In fig. 11.17 it is clear that controlling the rotations of the cylinders with sinusoidal functions can reduce the drag coefficient significantly. However,



(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

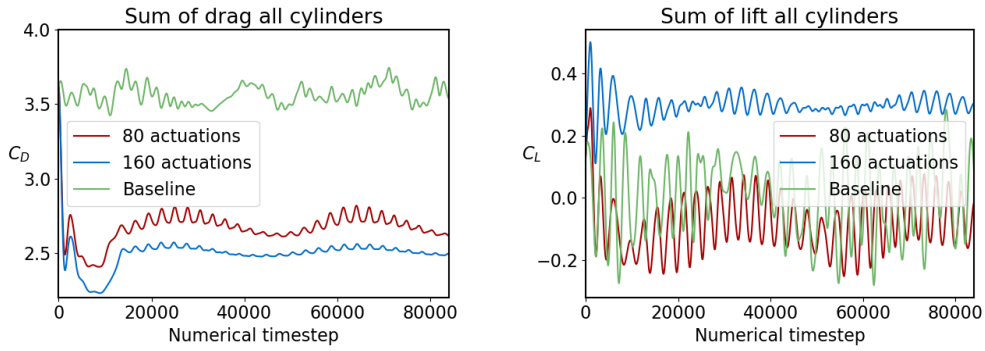
Figure 11.17: Drag reduction flow control by sinusoidal control functions compared to baseline flow. Sinusoidal control functions are capable of similar drag reduction as the DRL agents they are based on, but both strategies are unable to reach the stable oscillating flow of the 160 actuations DRL agent.

simulations at both $Re = 100$ and $= 150$ have shown that sinusoidal control functions are unable to establish a stable flow when reducing drag. Note that using sinusoidal control functions as we have corresponds to what is known as *open-loop* control. Meaning that no sensor readings or environment observations are required, and the control (rotations) will continue in the same manner independent of the state of the flow. In a real-life application this would be much easier to implement, as there would be no need for sensor readings or real-time computations to determine correct control actuations.

For $Re = 100$, in fig. 11.4, we observed that constant actuations were able to reduce the drag of the flow while avoiding larger fluctuations of the drag coefficient. In figure fig. 11.18 we plot the results for when we used similar control strategies for $Re = 150$ simulations.

The final control strategies applying constant rotations at $Re = 150$ are presented in fig. 11.18. It is once more clear that the control strategy based on the better DRL model performs better, reaching lower drag coefficient values, smaller oscillations, and a more stable lift coefficient. The constant control strategy based on the 80 actuations per episode DRL agent is also able to reduce drag significantly, but with less success than the actual DRL agent.

In table 11.3 we compare the drag and lift coefficient values of the different strategies, similarly to what we have done in the previous section of simulations at $Re = 100$.



(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

Figure 11.18: Drag reduction flow control by constant rotations compared to baseline flow. Constantly rotating the cylinders at fixed magnitude is again a very effective method of drag reduction. Compared to the sinusoidal control in fig. 11.17 the drag coefficient values are more stable, while at the same time obtaining similar average and minimum results. The lift is also more stable when applying constant actuations compared to sinusoidal control, especially for the strategy based on the 160 actuations agent. It does however, not reach the stable periodic oscillations observed for the DRL agent updating rotations 160 times per episode, which is a clear winner when it comes to drag reduction at $Re = 150$.

Control strategy	$C_D \pm$ std	$C_L \pm$ std	C_D reduction
Baseline	3.5724 ± 0.0619	-0.0128 ± 0.1293	0 %
DRL 80 actions	2.5342 ± 0.0612	-0.1099 ± 0.1403	29.06 %
DRL 160 actions	$\mathbf{2.4347 \pm 0.0212}$	$\mathbf{0.2511 \pm 0.1014}$	**31.85** %
Sinusoidal actions - 80	2.6938 ± 0.0765	-0.0784 ± 0.2734	24.59 %
Sinusoidal actions - 160	2.4874 ± 0.0734	0.2820 ± 0.0878	30.37 %
Constant actions - 80	2.6792 ± 0.0537	-0.0801 ± 0.0782	25.00 %
Constant actions - 160	2.5089 ± 0.0220	0.2963 ± 0.0180	29.77 %

Table 11.3: The final drag and lift coefficients of the different control strategies created to reduce drag at $Re = 150$ are presented. Each value is calculated as the mean value over the last half of a single run evaluation simulation. The control strategy we determine gave the best results is highlighted in bold. The drag reduction in % is calculated as $(C_{D,\text{baseline}} - C_{D,\text{control}})/C_{D,\text{baseline}}$.

11.2.2 Drag Increase

As fig. 11.19 clearly shows both agents learning to increase the drag also for $Re = 150$ we now want to investigate simulations without added exploration noise as we have done previously. The deterministic simulations for

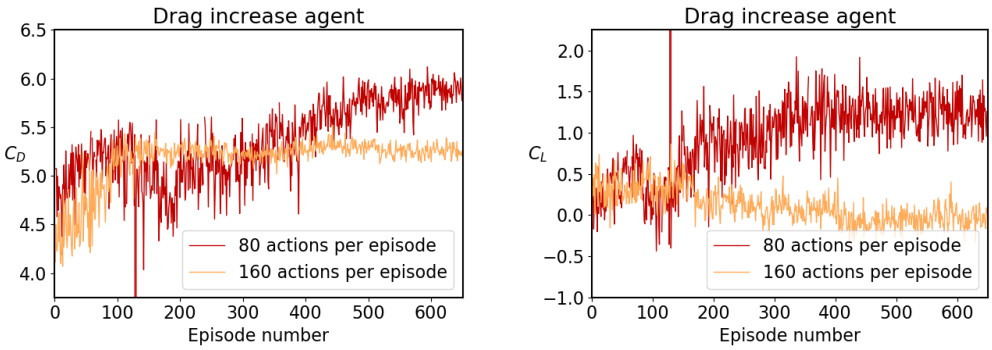
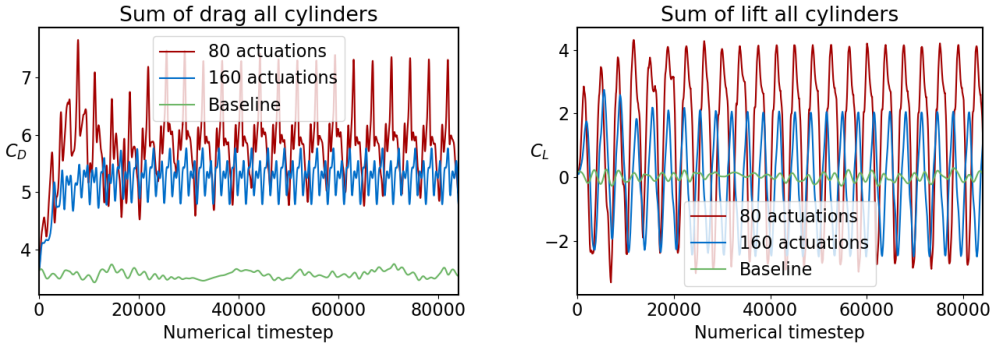


Figure 11.19: Drag and lift coefficient average per episode while training the DRL agent to increase the drag in the system. At $Re = 100$ we observed that both agents converged to very similar drag coefficient values, but for $Re = 150$ we can now see that the agent using 80 actuations per episode is seemingly converging to a significantly higher drag coefficient value. From subsection 11.2.2 we can also clearly see the strategy of the two agents clearly diverging where the average lift per episode diverges from zero-centered fluctuations.

increased drag using pre-trained DRL agents to increase drag are compared to the baseline simulation in fig. 11.20.

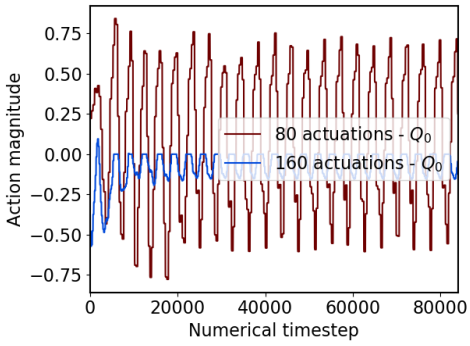
Both agents have learned effective strategies for increasing drag, but while previous simulations have shown quite similar results and strategies for both agents using 80 and 160 actuations per episode, we are now observing a significant difference between the two. In fig. 11.21 we compare the rotation inputs given to each cylinder for the two agents.

The sinusoidal control methods fitted to the 80 actuations agent are able to significantly increase the drag coefficient, to a similar degree as observed of the 160 actuations DRL agent. However, the sinusoidal control method fitted to the 160 actuations agent is not able to achieve the same level of drag increase, similar to what we observed of the sinusoidal control method based on the 80 actuations agent at $Re = 100$, see fig. 11.11a. The algorithmic fitting of sine functions to the actions taken by the 80 actuations agent showed some clear discrepancies, and we decided to try a manual tuning to better fit the sine functions to the known actuations of the DRL agent. Figure 11.22a shows us that the differences between the algorithmic fit, and the manually fitted control functions are insignificant, and have no impact on the effectiveness of sinusoidal control.

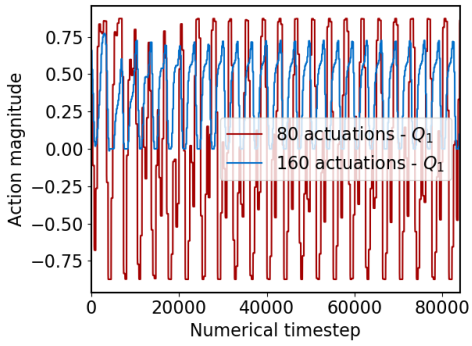


(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

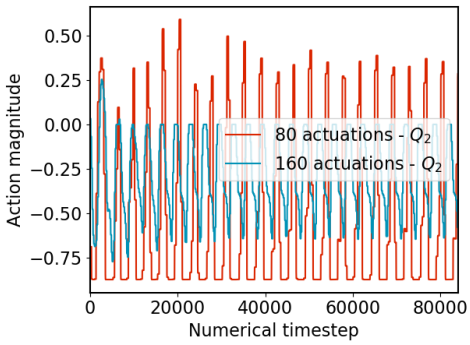
Figure 11.20: Drag increase flow control by DRL agents compared to baseline flow. As for $Re = 100$ the drag coefficient varies a lot more when we want to increase the drag compared to reducing drag. Especially the 80 actuations agent causes very large fluctuations of ± 1 centered around $C_D \approx 6$. An interesting thing to note is that while we observed that having more actuations per episode was a benefit when we wanted to reduce drag at $Re = 100$, we now observe that when we want to increase the drag at $Re = 150$ this is no longer beneficial. The baseline drag we are comparing the results with is the exact same simulation as compared to the DRL agents reducing drag, but due to the much larger range of values the variations we noticed previously seem insignificant compared to the fluctuations caused by applying control. Figure 11.20b shows a clear difference between the two agents. The 160 actuations agent is applying a strategy where the lift is zero-centered, although with quite large amplitude, while the control applied by the 80 actuations agent causes the lift to move from being zero-centered to oscillating while centered around ≈ 1 instead. Thus, if working with a more stable lift is desirable, then the DRL agent applying 160 actuations per episode might be better suited to the task.



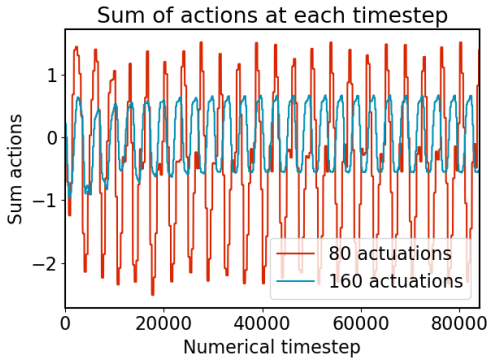
(a) Rotations applied to the first cylinder.



(b) Rotations applied to the top trailing cylinder.

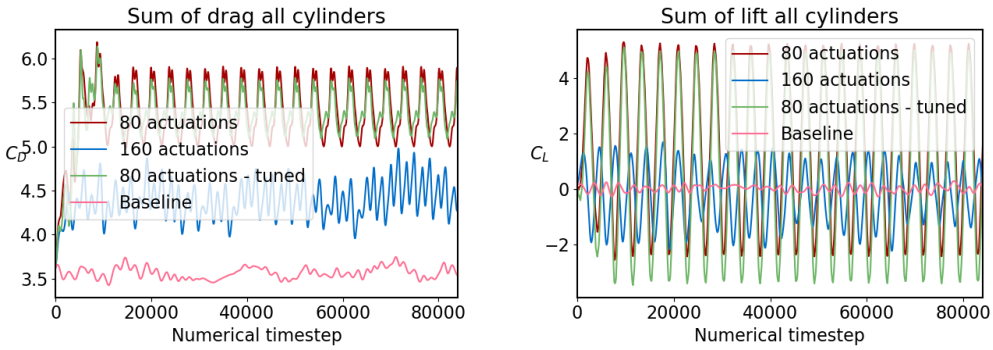


(c) Rotations applied to the bottom trailing cylinder.



(d) Sum of actions per numerical timestep.

Figure 11.21: Control actuations to increase drag by DRL agents. As was indicated in fig. 11.20b the control strategies differ significantly. Most obvious is the difference in rotations applied to the first cylinder where the 160 actuations agent uses very small rotations whereas the 80 actuations agent use oscillating rotations of high magnitude. For the agent using 160 actuations the trailing cylinders are only rotated in one direction or standing still, while the 80 actuations agent apply rotations to trailing cylinders in either direction and with slightly larger magnitudes. Figure 11.21d we have summed the rotation of all three cylinders at each timestep. If the sum of actions has a large magnitude the cylinder rotations are in large part moving in the same direction, while values closer to zero will mean that the rotations are moving counter to each other. For the 80 actuations agent the sum of actions magnitude is large in either direction, meaning that the oscillations seen in the rotations per cylinder are largely synchronised to work in the same direction at the same time. On the other hand the 160 actuations agent seems to be ignore using the first cylinder and then only apply rotation to one of the two trailing cylinders at a time.



(a) Drag coefficient vs. numerical timestep. **(b)** Lift coefficient vs. numerical timestep.

Figure 11.22: Drag increase flow control by sinusoidal control functions compared to baseline flow. The sinusoidal control functions are able to increase the drag to a similar degree as the 160 actuations DRL agent, but are not able to reproduce the result of the 80 actuations agent. This strongly indicates that the DRL agent has been able to learn an effective and complex control function that is not "easily" reproducible, and would be very difficult to find with no previous experience with the system. Note that the sinusoidal control functions fitted to the DRL agent of 160 actuations, are not able to increase the drag of the flow to the same degree as the two methods based on the 80 actuations agent. At $Re = 100$, in fig. 11.11a, we observed similar behavior of a sinusoidal control method, but that time it was based on the 80 actuations agent.

Control strategy	$C_D \pm$ std	$C_L \pm$ std	C_D increase
Baseline	3.5724 ± 0.0619	-0.0128 ± 0.1293	0 %
DRL 80 actions	$\mathbf{5.8799 \pm 0.5585}$	$\mathbf{1.2606 \pm 2.2487}$	**64.59 %**
DRL 160 actions	5.2953 ± 0.2423	-0.2277 ± 1.4531	48.23 %
Sinusoidal actions - 80	5.4222 ± 0.2975	1.4876 ± 2.6120	51.78 %
Sinusoidal actions - 80 (tuned)	5.4031 ± 0.1935	0.7457 ± 2.8759	51.25 %
Sinusoidal actions - 160	4.4227 ± 0.2004	-0.1838 ± 0.9164	23.80 %

Table 11.4: The final drag and lift coefficients of the different control strategies created to increase drag at $Re = 150$ are presented. Each value is calculated as the mean value over the last half of a single run evaluation simulation. The control strategy we determine gave the best results is highlighted in bold. The drag increase in % is calculated as $(C_{D,control} - C_{D,baseline})/C_{D,baseline}$.

11.3 Power spectral density (PSD)

According to Fourier analysis, a signal can be decomposed into a combination of discrete frequencies. A power spectral density plot can take a decomposed signal and describe which frequencies a signal consists of by measuring the intensity of each discrete frequency. We can treat our drag- and lift coefficient data as a signal and create PSD plots that can tell us about the significant frequencies present in the flow when control is applied or not. The following plots are all based on data of the total drag coefficient.

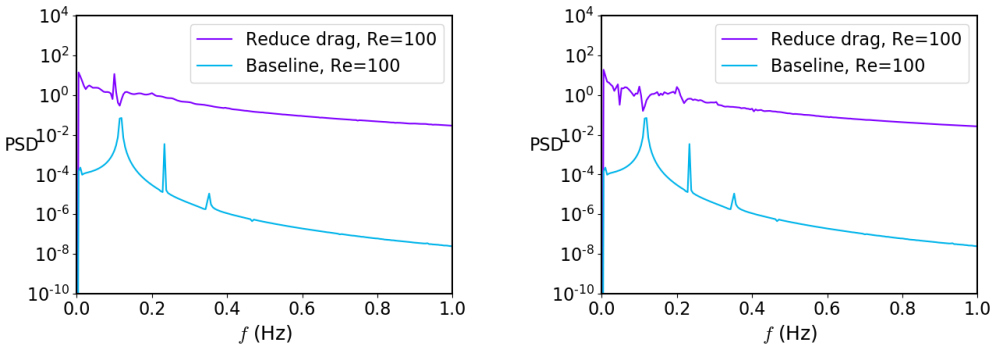
If the oscillations in drag can be fitted to a single frequency we can expect a PSD plot to contain a single, strong peak at the corresponding frequency. On the other hand, if the oscillations are more chaotic they are more likely to be the result of a combination of signals at different frequencies, and we will see multiple peaks in a PSD plot.

11.3.1 PSD of reducing drag agents

We start by creating PSD plots of the DRL agents that are trained to reduce the drag in the fluidic pinball system, and compare the PSD of the controlled flows with the PSD of the baseline flow without control. The comparison should give us a clear indication as to whether applying control interferes with the natural frequency of the system, or if the DRL agent controls the flow in such a way that the natural frequency is not affected.

In fig. 11.23 we can observe that the oscillations of the total drag of the baseline flow at $Re = 100$ is governed by three distinct frequencies. When we apply control with DRL agents we can see a clear effect on the governing frequencies, and the oscillations of the total drag are changed. For the first agent, in fig. 11.23a, we can see one clear peak at a frequency slightly lower than the main frequency of the baseline flow, and the two secondary peaks observed in the baseline are gone. The oscillations in the drag of the controlled flow is thus governed by a single frequency that is shifted from the baseline. The applied control is thus impacting the oscillations of the flow. The second agent does not show the same clear governing frequency as the first agent, indicating more complex oscillations of the total drag taking place, following a combination of frequencies. By looking at fig. 11.2 we can see that drag coefficient of the agent using 80 actuations per episode is very periodically stable and looks very much like a simple sinusoidal curve when stabilized. On the other hand, the agent of 160 actuations per episode oscillates in a more complicated manner, more reminiscent of how a combination of sinusoidal functions might interact.

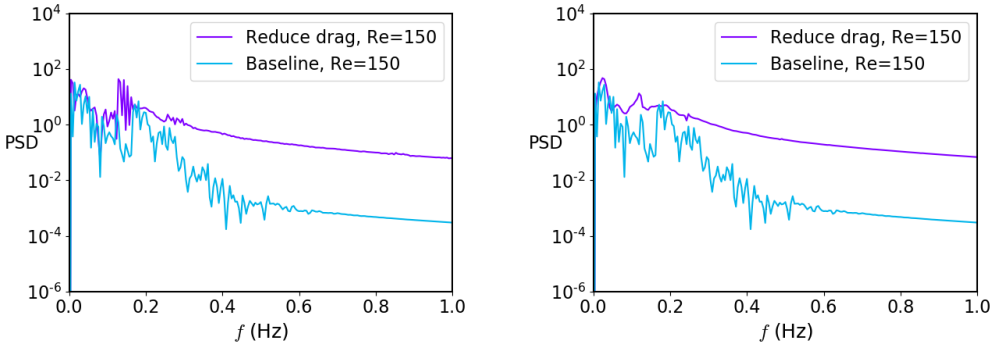
Comparing the baseline PSD of $Re = 150$, seen in fig. 11.24, with the baseline of $Re = 100$ significant differences become apparent. Where the baseline



(a) DRL agent of 80 actuations per episode. **(b)** DRL agent of 160 actuations per episode.

Figure 11.23: PSD plot of the total drag coefficient for the two DRL agents trained to reduce drag at $Re = 100$, compared to the PSD of the baseline flow. The baseline flow consists of three very distinct peaks, while the flows controlled by DRL agents does not have the same distinctive peaks. For the first agent we have a relatively clear peak for a frequency slightly lower than the largest peak of the baseline flow, and notice that the two secondary frequencies of the baseline are quite clearly gone. The second agent does not have a very clear peak, indicating that the oscillations in drag are not as dominated by a single frequency, but is rather a combination of several frequencies.

flow at $Re = 100$ is dominated by three distinct frequencies, the oscillations of the baseline drag at $Re = 150$ are significantly more complicated, and consists of a much larger variety of frequencies. If we look at the baseline drag coefficient in fig. 10.9b the baseline flow at $Re = 150$ varies a lot, and does not follow a simple sinusoidal curve as the baseline at $Re = 100$. With some goodwill we might say that strongly fluctuating peaks can be spotted around the same frequencies as for $Re = 100$, but it is apparent that the oscillations of the baseline drag at $Re = 150$ consists of more than a few separate frequencies. After we apply control the number of frequencies governing the drag oscillations are reduced significantly. The oscillations of the controlled flow using the agent applying 80 actuations per episode consists of a few close-laying frequencies. If we combine multiple sine functions which only vary by a small margin the resulting function will in large part behave similarly to the separate functions, but will eventually reach certain areas where the individual functions interfere with each other. If we look at the drag coefficient of the agent in fig. 11.15a we can see that the drag coefficient varies in a clearly sinusoidal fashion, but at the same time the oscillations are not as stable as can be seen of the second agent applying 160 actuations per episode.



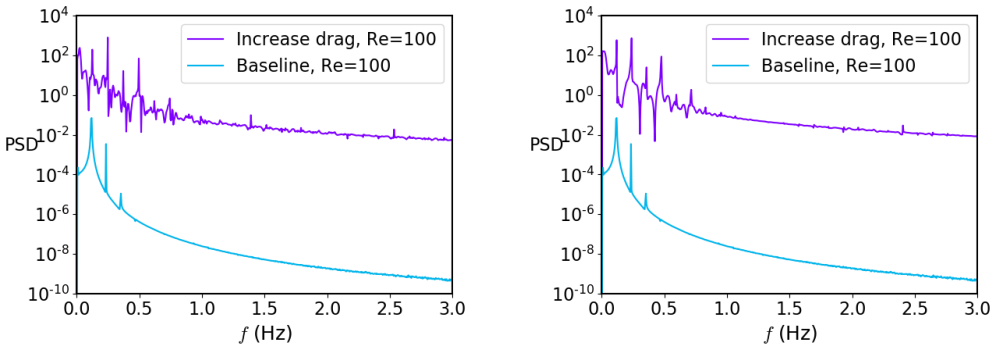
(a) DRL agent of 80 actuations per episode. **(b)** DRL agent of 160 actuations per episode.

Figure 11.24: PSD plot of the sum of drag coefficients for the two DRL agents trained to reduce drag at $Re = 150$. We should first note the significant difference between the baseline frequencies of $Re = 150$ compared to $Re = 100$. The reason behind is apparent if we look at fig. 10.9b where it is obvious that the baseline flow is quite chaotic, and not periodically oscillating as the baseline of $Re = 100$. The oscillations of the baseline at $Re = 150$ are governed by significantly more frequencies than the oscillations of the $Re = 100$ baseline. Both agents controlling the flow reduce the amount of governing frequencies by a significant amount. The oscillations in the resulting drag of the first agent are governed by a few near-laying frequencies. The oscillations in the drag coefficient of the second agent is largely controlled by a single frequency, but note that the peak is not as substantial as the comparable peaks, meaning that other frequencies will also have some impact.

11.3.2 PSD of increasing drag agents

Moving on to investigating the PSD plots of the DRL agents trained to increase the drag of the system we can expect to see more power per frequency due to the stronger oscillations present in the simulations for increasing drag. As in subsection 11.3.1 we start by looking at PSD plots at $Re = 100$, before moving on to the PSD plots of increasing drag at $Re = 150$.

In fig. 11.25 we observe that the DRL agents controlling the flow to increase the drag of the system causes an increased number of important frequencies, opposite of what we observed for drag reducing agents in fig. 11.23. The governing frequencies are also moved towards the higher frequencies rather than towards lower frequencies, indicating an increase in the number of oscillations of the drag coefficient. From the visualizations of the flow we know that the drag oscillated because of vortex shedding, and we can thus conclude that as the oscillations of the drag increase, so does the vortex shedding frequency. In fig. 11.8a we can see that the oscillations are very much larger than what is observed in the baseline flow, as well as the oscillations

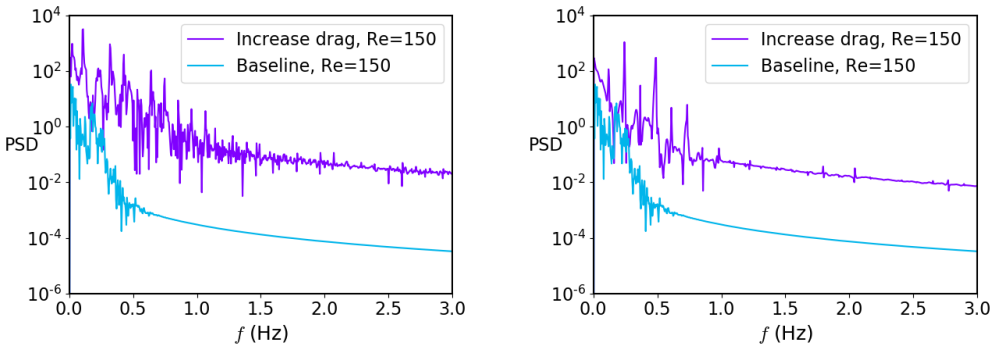


(a) DRL agent of 80 actuations per episode. **(b)** DRL agent of 160 actuations per episode.

Figure 11.25: PSD plot of the sum of drag coefficients for the two DRL agents trained to increase drag at $Re = 100$. Opposite of what we observed for reducing drag, the flows controlled by DRL agents see a large increase in significant frequencies. For the flow controlled by either agent the PSD consists of more peaks indicating significant frequencies, as well as a few clear negative peaks indicating that some frequencies are not present at all. The results of the control applied by the two agents are presented in fig. 11.8a, where we can see large oscillations in the drag coefficient. It is also apparent that the oscillations do not follow a nice and smooth sinusoidal curve as we observed for drag reduction agents, but are instead shaped quite sharply where every other oscillation is smaller.

of the drag coefficient when the flow is controlled by drag reducing agents. The oscillations of the drag increase agents are also not as smooth as what is observed during drag reduction, which can indicate that the oscillations in drag is controlled by more than a single frequency and that the overlapping of multiple sine functions can cause local interference causing the amplitude of the oscillations to vary.

For $Re = 150$ we observe that the oscillations are controlled by significantly higher frequencies than what we saw for $Re = 100$. The oscillations of the DRL agent applying 80 actuations per episode are governed by frequencies up to 1 Hz, compared to the other agents where such frequencies only have very minor impact. In fig. 11.20a we can observe the DRL agent, and the resulting drag coefficient of the applied control. The oscillations of the drag coefficient are very large compared to all the other agents we have observed, and also consist of multiple smaller oscillations between each large oscillation away from the mean. Figure 11.26a can give us an indication to the underlying frequencies of the oscillations. A great number of frequencies, also of higher frequency value, seem to have significant impact on the drag oscillation, indicating that the oscillations can be described as a combination of many sine functions with frequencies determined by the PSD plot.



(a) DRL agent of 80 actuations per episode. **(b)** DRL agent of 160 actuations per episode.

Figure 11.26: PSD plot of the sum of drag coefficients for the two DRL agents trained to increase drag at $Re = 150$. Where the PSD plots of the two DRL agents at $Re = 100$ are quite similar, we can see much more significant differences between the agents at $Re = 150$. The oscillations in drag of the agent applying 80 actuations per episode are governed by a wide range of frequencies, although with decreasing magnitude, as we have generally seen for the other plots as well. The drag coefficient oscillations observed of the agent using 160 actuations per episode consist of significantly fewer frequencies. We also note that where the oscillations of the second agent are grouped closer, and consist of lower frequencies than the relatively high frequencies of the first agent. However, by comparing either agent to the baseline oscillations it is clear that neither agent reduces the oscillation frequency, but are rather increasing the frequency, just as we saw for $Re = 100$.

With many sine functions of significantly different frequencies we would observe serious interference between functions, which would result in smaller oscillations when the interference is destructive, and large oscillations when many of the functions experience positive interference. Figure 11.26b is more similar to what we saw in fig. 11.25. This understanding is reinforced if we compare the drag coefficient oscillations of the 160 actuations at $Re = 150$ in fig. 11.20a to the oscillations of the two $Re = 100$ agents in fig. 11.8.

Part IV

Conclusion and Discussion

Chapter 12

Conclusion

12.1 Summary

After having explained and discussed a wide variety of theory, methodology, simulations, and results, it is convenient to briefly summarize the thesis. We started by placing the work of the thesis in the overlap of machine learning in fluid dynamics, and optimization in fluid dynamics. More specifically we wanted to apply deep reinforcement learning (DRL) on the active flow control (AFC) problem of the *fluidic pinball*.

In part I important theoretical background was presented. In chapter 2 we gave a thorough introduction to some of the key elements of machine learning. Starting with a relatively low-level introduction of learning algorithms in general, before moving on to data fitting, regularization, hyperparameters, and gradient descent methods. Chapter 3 presented the key components of deep learning, some of which have been implemented in later chapters. Machine learning and deep learning are closely related terms, and the field of deep learning is often considered to be a part of the very wide umbrella term, *machine learning*. Another area covered by machine learning is reinforcement learning (RL), which was presented in chapter 4, where we focused on giving the reader an understanding of what reinforcement learning is, and then presented the proximal policy optimization algorithm.

Moving on to chapter 5 we introduced active flow control (AFC), starting with classical methods like linear control and gradient-based methods, before we presented deep reinforcement learning (DRL) methodology as an optional method of active flow control. Chapter 6 gave brief summaries of the two journal articles, Rabault et al. [33] and Rabault and Kuhnle [32], which introduced the methodology implemented in the thesis.

In part II the technical necessities along with the most important parts of the implemented code were presented. Chapter 7 presented important software like Docker, along with the key Python packages that were used

in the implementation. The numerical solver of the Navier-Stokes equations was presented in chapter 8, and the chapter served the dual purpose of also presenting key elements of the finite element method (FEM). Chapter 9 presented selected parts of the implemented code related to combining deep reinforcement learning (DRL) methodology with the previously presented numerical flow solver. In chapter 10 the system to be simulated was presented, along with necessary preparations like mesh refinement and flow initialization.

In part III and chapter 11 the results of the simulations described in chapter 10 were presented. Starting with simulations at $Re = 100$ and then $Re = 150$ different flow control strategies were presented and discussed. A power spectral density (PSD) analysis of the deep reinforcement learning control strategies was also presented, giving insight to how the frequency of vortex shedding was altered by active flow control.

12.2 Discussion

12.3 Future work

Appendices

Acronyms

Re Reynolds number

AFC active flow control

AI artificial intelligence

ANN artificial neural network

CNN convolutional neural network

CPI conservative policy iteration

DNN dense neural network

DRL deep reinforcement learning

FC flow control

FCNN fully connected neural network

FEM finite element method

FFNN feedforward neural network

FM fluid mechanic

GD gradient descent

IPCS incremental pressure correction scheme

LQE linear quadratic estimation

LQG linear quadratic Gaussian

LQR linear quadratic regulator

LReLU leaky rectified linear unit

ML machine learning

MLP multilayer perceptron

MPI message passing interface

MSE mean squared error

NN neural network

PIV particle image velocimetry

PPO proximal policy optimization

PSD power spectral density

ReLU rectified linear unit

RL reinforcement learning

RNN recurrent neural network

ROM reduced order model

SGD stochastic gradient descent

TRPO trusted region policy optimization

vCPU virtual CPU

VM virtual machine

References

[1] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization". In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305. ISSN: 15324435.

[2] Steven L. Brunton and Bernd R. Noack. "Closed-loop turbulence control: Progress and challenges". In: *Applied Mechanics Reviews* 67.5 (2015). ISSN: 00036900. DOI: 10.1115/1.4031175.

[3] Varun Chandola, Arindam Banerjee, and Vipin Kumar. "Anomaly Detection: A Survey". In: *ACM Computing Surveys* 41.3 (2009). ISSN: 0360-0300. DOI: 10.1145/1541880.1541882.

[4] S. Scott Collis et al. "Issues in active flow control: Theory, control, simulation, and experiment". In: *Progress in Aerospace Sciences* 40.4-5 (2004), pp. 237–289. ISSN: 03760421. DOI: 10.1016/j.paerosci.2004.06.001.

[5] Nan Deng et al. "Low-order model for successive bifurcations of the fluidic pinball". In: (2018). arXiv: 1812.08529. URL: http://arxiv.org/abs/1812.08529.

[6] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12 (2011), pp. 2121–2159. URL: http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[7] Thomas Duriez, Steven L. Brunton, and Bernd R. Noack. *Machine Learning Control – Taming Nonlinear Dynamics and Turbulence*. Springer International Publishing, 2017. ISBN: 978-3-319-40624-4. DOI: 10.1007/978-3-319-40624-4. URL: https://www.springer.com/gp/book/9783319406237.

[8] Paul Garnier et al. "A review on Deep Reinforcement Learning for Fluid Mechanics". In: (2019). arXiv: 1908.04127. URL: http://arxiv.org/abs/1908.04127.

[9] Christophe Geuzaine and Jean François Remacle. "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *International Journal for Numerical Methods in Engineering* 79.11 (2009), pp. 1309–1331. DOI: 10.1002/nme.2579. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2579.

[10] Katuhiko Goda. "A multistep technique with implicit difference schemes for calculating two- or three-dimensional cavity flows". In: *Journal of Computational Physics* 30.1 (1979), pp. 76–95. ISSN: 10902716. DOI: 10.1016/0021-9991(79)90088-3.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: https://www.deeplearningbook.org/.

[12] Song Han et al. "Learning both weights and connections for efficient neural networks". In: *Advances in Neural Information Processing Systems* 2015-Janua (2015), pp. 1135–1143. ISSN: 10495258. arXiv: 1506.02626.

[13] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 08936080. DOI: 10.1016/0893-6080(89)90020-8.

[14] Anil K. Jain, Jianchang Mao, and K. M. Mohiuddin. "Artificial neural networks: A tutorial". In: *Computer* 29.3 (1996), pp. 31–44. ISSN: 00189162. DOI: 10.1109/2.485891.

[15] Sham Kakade and John Langford. "Approximately Optimal Approximate Reinforcement Learning". In: *International Conference on Machine Learning*. 2002, pp. 267–274. URL: https://homes.cs.washington.edu/%7B~%7Dsham/papers/rl/aoarl.pdf.

[16] Diederik P. Kingma and Jimmy Lei Ba. "Adam: A method for stochastic optimization". In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings* (2015), pp. 1–15. arXiv: 1412.6980.

[17] Alex Kriezhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. 2012, pp. 1097–1105. ISBN: 9781420010749. DOI: 10.1201/9781420010749.

[18] Alexander Kuhnle, Michael Schaarschmidt, and Kai Fricke. *Tensorforce: a TensorFlow library for applied reinforcement learning*. 2017. URL: https://github.com/tensorforce/tensorforce.

[19] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python: The FEniCS Tutorial*. Vol. I. 2017, p. 153. ISBN: 0027-8424. DOI: 10.1073/pnas.202491499. URL: http://www.springer.com/series/13548%7B%5C%%7D0Ahttp://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%7B%5C&%7Dpath=ASIN/3319524615.

[20] Moshe Leshno et al. "Multilayer feedforward networks with a nonpolynomial activation function can approximate any function". In: *Neural Networks* 6.6 (1993), pp. 861–867. ISSN: 08936080. DOI: 10.1016/S0893-6080(05)80131-5.

[21] Anders Logg, Kent Andre Mardal, and Garth N. Wells. *Automated solution of differential equations by the finite element method*. 2012, pp. 77–94, 399–440. ISBN: 9783642230981. DOI: 10.1007/978-3-642-23099-8_1.

[22] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (1943), pp. 115–133. ISSN: 00074985. DOI: 10.1007/BF02478259.

[23] Pankaj Mehta et al. "A high-bias, low-variance introduction to Machine Learning for physicists". In: *Physics Reports* 810 (2019), pp. 1–124. ISSN: 03701573. DOI: 10.1016/j.physrep.2019.03.001. arXiv: 1803.08823. URL: https://doi.org/10.1016/j.physrep.2019.03.001.

[24] Chulhong Min and Haecheon Choi. "Suboptimal feedback control of vortex shedding at low Reynolds numbers". In: *Journal of Fluid Mechanics* 401 (1999), pp. 123–156. DOI: doi:10.1017/S002211209900659X.

[25] Tom M. Mitchell. *Machine Learning*. New York: McGraw-Hill, 1997. ISBN: 9780071154673.

[26] Frank Muldoon. "Control of hydrothermal waves in a thermocapillary flow using a gradient-based control strategy". In: *International Journal for Numerical Methods in Fluids* 72 (2013), pp. 90–118. DOI: 10.1002/fld.373.

[27] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. Omnipress, 2010, pp. 807–814.

[28] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com/index.html.

[29] Georgios Pechlivanoglou. "Passive and active flow control solutions for wind turbine blades". In: *PhD thesis* (2012). DOI: 10.14279/depositonce-3487.

[30] Ludwig Prandtl. "Über Flussigkeitsbewegung bei sehr kleiner Reibung". In: *Verhandl. III, Internat. Math.-Kong., Heidelberg, Teubner, Leipzig, 1904* (1904), pp. 484–491.

[31] Jean Rabault and Alexander Kuhnle. "Accelerating Deep Reinforcement Learning of Active Flow Control strategies through a multi-environment approach". In: (2019). arXiv: 1906.10382. URL: http://arxiv.org/abs/1906.10382.

[32] Jean Rabault and Alexander Kuhnle. "Accelerating Deep Reinforcement Learning of Active Flow Control strategies through a multi-environment approach". In: *Physics of Fluids* 31.9 (2019), p. 094105. DOI: 10.1063/1.5116415. arXiv: 1906.10382. URL: http://arxiv.org/abs/1906.10382%20https://doi.org/10.1063/1.5116415.

[33] Jean Rabault et al. "Artificial neural networks trained through deep reinforcement learning discover control strategies for active flow control". In: *Journal of Fluid Mechanics* 865 (2019), pp. 281–302. ISSN: 14697645. DOI: 10.1017/jfm.2019.62.

[34] Benjamin Recht et al. "Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent". In: *Advances in neural information processing systems*. Curran Associates, Inc., 2011, pp. 693–701. DOI: 10.1016/s1350-4789(16)30126-x.

[35] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: (2016), pp. 1–14. arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

[36] John Schulman et al. "Trust Region Policy Optimization". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. 2015, pp. 1889–1897. URL: http://proceedings.mlr.press/v37/schulman15.html.

[37] John Schulman et al. "Proximal Policy Optimization Algorithms". In: (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[38] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning An Introduction*. 2nd ed. Cambridge, MA: MIT Press, 2018. ISBN: 9780262039246. URL: http://incompleteideas.net/book/the-book-2nd.html.

[39] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[40] Kristian Valen-Sendstad et al. "A comparison of finite element schemes for the incompressible Navier–Stokes equations". In: (2012), pp. 399–420. DOI: 10.1007/978-3-642-23099-8_21.

[41] Stéfan Van Der Walt, S. Chris Colbert, and Gaël Varoquaux. "The NumPy array: A structure for efficient numerical computation". In: *Computing in Science and Engineering* 13.2 (2011), pp. 22–30. ISSN: 15219615. DOI: 10.1109/MCSE.2011.37. arXiv: 1102.1523.

[42] Mitchell M. Waldrop. "More than moore". In: *Nature* 530.7589 (2016), pp. 144–147. ISSN: 0028-0836. DOI: 10.1038/530144a. URL: https://doi.org/10.1038/530144a.