VRIJE
UNIVERSITEIT
BRUSSEL

von KARMAN INSTITUTE
FOR FLUID DYNAMICS

REINFORCEMENT LEARNING FOR ACTIVE FLOW CONTROL

Desmet Mitja

October 10, 2020

Promotor: Dr. Miguel Alfonso Mendez
Supervisor: Fabio Pino
von Karman Institute for fluid dynamics

Abstract

We investigate the use of reward shaping to speed up the learning process of reinforcement learning agents for active flow control. We begin with an overview of the field of reinforcement learning and introduce the concepts which lie at its foundations. We present the idea of reward shaping and propose different techniques to apply it inside a flow control problem. The approaches are evaluated in an environment governed by the advection equation. We compare the best models learned over two different numbers of iterations and conclude that while reward shaping does not improve on the best learned model, it reduces the amount of steps required to find it.

Contents

List of Symbols

Acronyms

ML	Machine learning
RL	Reinforcement learning
NN	Neural network
DNN	Deep neural network
SGD	Stochastic gradient descent
MDP	Markov decision process
QL	Q-learning
DQN	Deep Q Neural Network
PBRS	Potential-based reward shaping
PPO	Proximal policy optimizer
BRS	Belief reward shaping
RND	Random network distillation
VKI	von Karman Institute

Chapter 1

Introduction

Flow control is a major field of fluid dynamics that has a considerable interest both in academic and industrial perspectives. Some prominent applications go from the reduction of drag on air plane wings to the control of heat flux on the external surface of spacecrafts during the re-entry phase [1]. Research is done with both passive and active solutions for flow control. While passive control mechanisms tend to be easier to implement, there is a risk of them interfering with the rest of the system, because they are always doing something. Typically, they also cannot deal with a broad range of conditions, but are rather more specific to a certain situation.

Active flow control mechanisms on the other hand are designed to respond to the external conditions. These are thus potentially much more versatile. They activate only when required, hence they do not interfere with the natural performance and do not hold potential resources away from the system. An example in everyday life of active flow control are the latest generations of headphones, which are equipped with active noise cancellation. While noise cancellation is a little more subtle than what we usually understand under flow control, the idea in both cases is to cancel out (some) waves.

Given its importance in performance improvement, a lot of effort is put into finding optimal ways to control flows. On the theoretical side, not a lot of progress is made, mainly due to the lack of closed-form solutions to the fluid equations and the infinite state space dimension of most fluid dynamics problems. Therefore researchers are trying to tackle the problem using experimental tools instead. As a result of the increased precision of the equipment, more can be learned from the experiments. The drawback is these experiments are usually expensive and take up a lot of space, as they require large-scale facilities.

Although it has been an active research topic for decades, flow control is going through a blooming season owing to the rise of machine learning control techniques. More in particular the field of reinforcement learning has given proof to be a very promising method. It has recently emerged as an important domain in machine learning, as a result of different important breakthroughs. Currently, most research on the topic involves games, as they provide a well-defined environment in which a computer can learn. However, using the right tools, the ideas of reinforcement learning can be extended to any problem.

1.1 Aim of the project

This research project aims at investigating the control of a linear representative of more complex phenomena test cases. First we present a short overview of the main machine learning concepts in Chapter 2. We also introduce the ideas at the foundation of reinforcement learning together with some basic algorithms. We end the Chapter by looking at the algorithm we use in our research and by presenting a state-of-the-art reward shaping method we consider when tackling our control problem.

The environment we consider in Chapter 3 is governed by the advection equation, which is linear and allows for an analytic control solution for a simple harmonic perturbation. Therefore we know the optimal control action, which allows us to accurately measure the performance of our algorithms. There exist many different strategies in reinforcement learning and it is not obvious which one is best suited to our problem. The goal will be to find an efficient algorithm, which manages to find the optimal control action in the least amount of time.

The use of reinforcement learning for active flow control was first proposed in [2]. Since then, more research is going into tackling new problems with reinforcement learning, from driving cars to protein folding [3]. More recently, researchers at the von Karman Institute for fluid dynamics have tried to apply newly developed techniques from reinforcement learning to active flow control [4]. This project is a continuation of that and will contribute to the future development of these ideas.

Acknowledgements I would like to thank prof. Miguel Mendez for giving me the opportunity to have done this project, even if the circumstances were less than ideal. I also owe a lot to Fabio Pino, who guided me during this project and provided me with the material to learn from. I hope this project can help him in return, during his PhD. Lastly, I should also thank Lorenzo Schena and Antonin Raffin for their help with the implementation of the code.

Chapter 2

Reinforcement learning

The journey in search for Artificial (general) Intelligence, AI for short, has been long and exciting, bringing us new insights on the nature of intelligence itself. It challenged humanity to rethink what intelligence really means. One remarkable example is the victory of AlphaGo over formerly world champion Lee Sedol in 2016 [5]. AlphaGo was designed by Deepmind, a company driving a lot of the innovations in the field of reinforcement learning (RL). The basic principle of RL is to create an *agent* which *learns* how to achieve a certain goal without human guidance. The agent does this by interacting with the environment and observing the reaction from the environment. Based on this reaction it will not only choose what action to take next, but most importantly evaluate its previous decisions. This last part is where the learning comes in and is what gives RL algorithms there power and versatility. After interacting with the environment many times, the agent will (hopefully) have learned an optimal way to achieve its goal.

We first go over some basic principles of machine learning in Section 2.1, focusing on the field of supervised learning. This leads us to the introduction of artificial neural networks as models of the decision making process. Then in Section 2.2 we look at reinforcement learning, another important field in machine learning. After discussing some key features, we go over some basic reinforcement learning algorithms in Section 2.3. In Section 2.4 we then present the algorithm that we used in our research. We will see in Section 2.5 that we can try to speed up the convergence of the algorithms by introducing some prior knowledge trough intermediate rewards. To conclude, in Section 2.6 we introduce a recently proposed reward shaping method that seems very promising and that we considered during this project.

2.1 Machine learning

The use of machine learning (ML) in different fields of science and society is not new. Computers can process large amounts of data and easily recognize patterns in it. The most basic use of ML is to classify data is some way or another. There are different ways of approaching this task, but most of them can be put into one of the following categories: supervised and unsupervised learning.

In the first case, the computer is given a set of training samples which are correctly labelled. The algorithm then looks at the training samples and constructs for itself a model how to classify the samples into the given categories. After training, the algorithm

can then look at new inputs and predict in which category they belong. In the case of unsupervised learning there is no set of labelled samples, there are even no known categories. The algorithm just tries to look at data and divide it into clusters with similar properties.

While unsupervised learning can be useful, most of the applications require the use of supervised learning. The goal is to understand what features are important to make a decision, whether it is deciding what animal is in a picture or for how much you can sell your house. We quantify this by using a hypothesis function, which has the form

$$h_\theta(x) = \theta^T x, \tag{2.1}$$

where x is a column vector containing all features we want to consider and θ represents the weights of these features. The relation does not have forcibly to be linear, it could be also non-linear, e.g. considering the product of one of more features. To give a concrete example, when estimating the worth of a house, some observables are for example the area of the house (x_1) and the number of rooms (x_2). In this case, one could consider the feature

$$x_1 x_2,$$

to capture more complicated desires, e.g. when someone wants a large house, he or she probably also wants a lot of different rooms.

By using the set of labelled inputs, the algorithm can adjust the weights θ to match the correct outcomes, in a process we call learning. Typically, learning is the slowest part of the operation. The algorithm needs a lot of time to adjust all the weights and see how it can come closer to the correct prediction. This gets more complicated the more features we want to consider. There is therefore an important balance here: on one hand, we would like to be able to use the algorithm to classify new inputs as fast as possible, but on the other hand it needs to do this with an acceptable accuracy. The more time it has to learn, the more accurate it will become. The accuracy is captured in the cost function $J(\theta)$, which quantifies how close the algorithm is to predicting the correct output. We measure the algorithms performance using a set of test examples, which are separated from the training set with which the algorithm learns. The exact form of the cost function depends on the problem, but one common form is

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2, \tag{2.2}$$

with m the number of samples in the test set and $y^{(i)}$ the correct label of sample i. Gradient descent (GD) is one of the oldest optimization algorithms to learn the weights θ and optimizes them given the correct outputs. GD tries to find a local optimum by going down (if we are looking for a minimum) the gradient of the cost function w.r.t. the weights. GD converges under quite general conditions [6], but tends to be quite slow.

The weight vector θ can be a matrix and in this case the hypothesis function outputs a column vector. Usually, the entries are interpreted as the probabilities the input belongs to the different classes. As a result, the algorithm can make a decision based on the hypothesis function output. When the number of features grows very large,

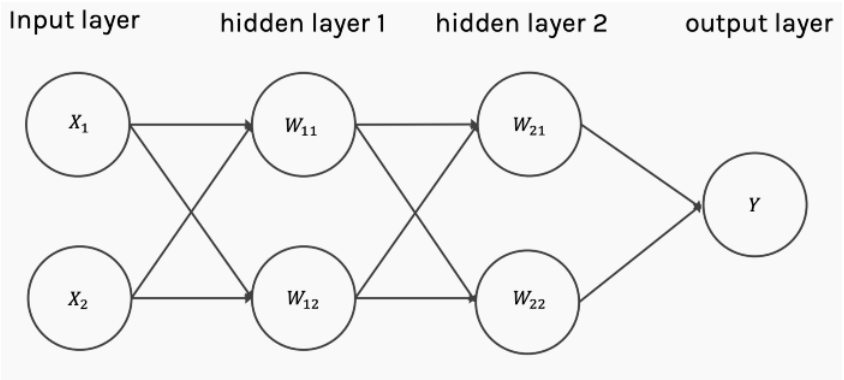

Figure 2.1: Example of a simple NN. Figure taken from [7].

the hypothesis function becomes very complex. This is where neural networks (NN) come in. They can approximate complex, non-linear functions and allow for an efficient optimization of the weights. On a basic level, a NN consists of layers of neurons. Each individual neuron receives an input, applies an activation function to that input, resulting in an output. However, the most general architecture allows for each layer to have a *bias unit*, which outputs a constant value and receives no input. For the first layer, the inputs are just the features we consider. The neurons in the other layers take as input a linear combination of all the outputs of the previous layer, to which they also apply the activation function. Each neuron is allowed to have its own weights for the linear combination. An example of a simple NN is shown in Figure 2.1.

To conclude, let us quickly introduce some terminology about NN. The first layer is called the input layer, as it takes the inputs we feed to the algorithm. These inputs are then forward propagated trough the network, via the hidden layers. These are all the layers between the first and the last one. A NN with more than 1 hidden layer is often referred to as a deep NN (DNN). The last layer then is called the output layer. NN can learn through a method which is called back-propagation. It allows for the information from the cost function to flow backwards through the network, making it possible to adjust all the different weights in the network. A detailed account on this topic can be found in section 6.5 of [8].

2.2 The goal of reinforcement learning

By now, ML has moved beyond supervised and unsupervised learning. One type of ML which has become very popular in the last two decades is reinforcement leaning. It differs significantly from the two previous mentioned paradigms, most notably in the fact that there is no pre-labelled data. Instead, the data is generated by the algorithm trough interactions with the environment. The active nature implies that decisions are made while data is flowing in and that these decisions also influence what data comes next. This captures the idea of incomplete information, as we face in real-world problems.

In RL, we focus on the agents instead of the algorithms which the agents use to learn. An agent has the ability to interact with its environment, which it might or might not be familiar with. Interactions happen through actions of the agent. These actions

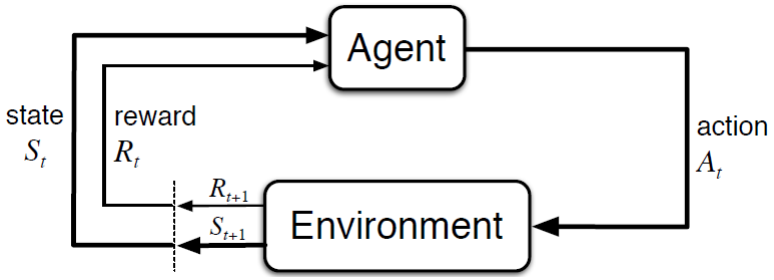

Figure 2.2: The agent-environment interaction in RL. Figure taken from [9].

influence the environment and the relation of the agent with respect to the environment. A good example to think of is a robot moving in a room. Let us say it can move left, right, forwards and backwards. In this case, the environment is not much influenced by the actions, but they will change the position of the robot in the room. If we now allow the robot to pickup or drop objects inside the room, then the environment can be substantially altered by the robots actions. The goal of the agent is to maximize something we call the *reward*. Rewards are given as the agent takes actions and can be either positive or negative. Trough learning, the agent will try to find how it accumulates as much of it as possible. The agent is essentially steered towards the intended behaviour, but without any explicit instructions. Taking again our example of a robot in a room, imagine we want it to go from point A to point B as quickly as possible. To incentivize it to go straight to its destination, we can give it a reward of -1 every time it takes a step. Detours will decrease the total reward it gets in the long run, so over time, it will learn to avoid making them.

The standard mathematical description of a problem in RL is a (finite) Markov decision process (MDP), which we will define in a moment. The agent-environment interaction of an MDP is summarised in Figure 2.2. The agent starts in a certain state S_t and chooses to take action A_t. This is the reason for the word *decision* in MDP: the agent has some agency about how it interacts. The word *Markov* in MDP refers to the property that the current state fully characterizes the process, thus there is no memory involved. The environment then responds to the action by feeding the agent with a new state S_{t+1} and a reward R_{t+1}. This reward tells the agent how good or bad it is to be in this new state. Based on this information, the agent can then make a decision what action to take next, but more importantly, it can also evaluate its previous decision. The subscripts refer to time steps, as we think of this processes as happing while time evolves.

To formalize this, we will mostly follow the conventions in [9], which is an excellent reference on anything RL related. A finite MDP is described by a tuple $\mathrm{M} = (S, A, P, R, \gamma)$, where S is a finite set of states the agent can be in and A is the finite set of all actions the agent can take. Note that generally $A(s)$ with $s \in S$, i.e. not all actions may be available in a certain state. When the agent takes action a in state s, it transitions to state s' with a probability described by $P : S \times A \times S \mapsto [0, 1]$. Together with the new state, the agent also receives a reward given by the reward function $R : S \times A \times S \mapsto \mathbb{R}$. In most applications R is a deterministic function, but more generally it can be a distribution. The last element of the tuple is $\gamma \in [0, 1]$, the discount

factor. More than merely a mathematical convenience, it indicates how much we care about future rewards.

Our agent is trying to maximize the total cumulative reward it receives over time. This idea is formulated in the reward hypothesis, as defined on p.53 in [9]:

The reward hypothesis. *All of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).*

To define this objective formally, we first define the discounted return as

$$G_t \equiv R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. \tag{2.3}$$

Here we see the discount factor coming in. It dictates the present value of future rewards: a reward is valued more, the more immediate it is. Moreover, the discount factor also bounds the infinite sum in Equation (2.3). At every time step, the agent has to choose its action in order to maximize the expected discounted return. The return, and therefore also the decision process, depends on the current state. The mapping from states to probabilities of selecting actions is called a policy. If the agent follows a certain policy π at time t, than $\pi(a|s)$ is the probability that it will select action a when it finds itself in state s. Given a policy π, we can define the value function of a state s by

$$v_\pi(s) \equiv \mathbb{E}_\pi[G_t|S_t = s], \forall s \in S. \tag{2.4}$$

Here t can be any time step and $\mathbb{E}_\pi$ denotes the expectation value if the agent follows the policy from there onward. This allows us to assign values to states and the agent will then be able to try reaching the states with the highest value. Similarly we have the action-value function for policy π

$$q_\pi(s, a) \equiv \mathbb{E}_\pi[G_t|S_t = s, A_t = a], \forall s \in S, a \in A. \tag{2.5}$$

Here the action a is chosen in state s, after which the policy π is followed. The value function and action-value are closely intertwined, but most of the times the latter is the more useful one. When an agent is in a state s and wants to decide which action it takes next, it will try to take one that maximizes the return. Another way of saying this, is that it will take the action which has the highest action-value in that particular state. Of course, that action will determine in which state the agent will end up in and that state has a value associated to it. Here we see the close interplay between the two functions, which is expressed in the equations

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s, a) \tag{2.6}$$

$$q_\pi(s, a) = \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]. \tag{2.7}$$

Using what we know, we can reformulate the RL problem. What we are looking for, is an optimal policy π_*, that maximizes the (action-)value function:

$$v_*(s) = \max_\pi v_\pi(s) \tag{2.8}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a). \tag{2.9}$$

Although there might be more than one optimal policy, it can be shown that they share the same optimal value and action-value function. The optimal policy represents the best way to move trough our environment, to achieve the goal as best as possible. Combining Equations (2.6) and (2.7) for π_*, we find the Bellman optimality equations for

$$v_*(s) = \max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_*(s')\right] \tag{2.10}$$

and

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right]. \tag{2.11}$$

Finite MDPs have a unique solution for these sets of equations, which immediately yield one optimal policy. Namely, we could select the action with the highest action-value in each state. Unfortunately, solving these directly is in practice a difficult task. Another approach is to estimate the values of the optimal (action-)value function from experience and learn them over time.

2.3 From exploring to learning

An RL agent is simply a program which is trying to maximize the cumulative reward it receives from the environment. According to the reward hypothesis, we can formulate each problem as a reward function which pushes the agent towards the goal we have in mind. As it turns out, designing a suitable reward function is not easy and we will touch on this subject in Section 2.5. Computers are very good at behaving exactly as you say and this means that we have to be very careful and explicit when implementing reward functions. A good example of this, was an experiment where an agent had to learn to go from point A to point B. In order to push the agent towards B, the researchers gave it a reward every time it approached B. However, there was no penalty for going away from it. So, what the agent found was that it could keep going in circles, accumulating rewards every time it moved closer towards B.

The first time an agent is initiated, it has only little information: the state it is currently in and the possible actions it is allowed to take in that particular state. For the first step, it has to take a random action. After taking that action, it receives a reward from the environment. This reward provides the agent feedback, informing it to how well the action was. Over time, with enough feedback, our agent can build up an understanding of the environment and find an optimal way to navigate it.

One of the most basic RL algorithms is Q-learning (QL). In QL, the agent keeps a table, the so-called Q-table, where it tracks its estimates of the action-value function for each action, in each state. As an example, consider a small gridworld as drawn in Figure 2.3. The agent start in the cell with S and has to find a way to G. If at any point in

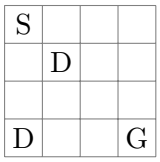

Figure 2.3: Small gridworld. Agent starts at S and its goal is to find G. If it hits a cell with D, it is penalized and send back to S.

time it hits a cell with D in it, it is game over. In this case it receives a large negative reward and the game is reset. At every step, the agent can either do nothing or move up, down, left or right. If it attempts to step off the map, nothing happens and it stays where it is. In order to make the agent go as fast as possible, we give a reward of -1 every time it takes an action (even if that action does not change its state).

In the gridworld example, the agent can be in any of the 16 possible states and in every state it can make 5 possible moves. Using Q-learning, the agent would have an internal 16×5 table, with as entries its own estimates for the *q-values*, which are nothing else than the values of the action-value function. The table starts with all zeros. At first, the agent has to make a random guess. This is the exploration phase. But, every time it receives a reward, it can judge how well it was to take that action in that particular state. It updates its Q-table using the Bellmann equation for going from state s to state s' via action a:

$$Q_{new}(s,a) = Q(s,a) + \alpha \left[R(s,a) + \gamma \cdot \max_{a'} Q'(s',a') - Q(s,a) \right]. \tag{2.12}$$

Here α is a parameter called the learning rate and γ is the discount factor we encountered before. The learning rate decides how quickly the agent abandons its former estimate. Formally it sets the step size in the gradient descent process. Usually this value is close to, but not exactly 1.

After a while, the estimates in the Q-table converge towards the true optimal q-values q_*. Once the agent gets to that point, it may start to favour some actions above others in certain states, as it knows they are likely to yield a higher cumulative reward. This is the beginning of the exploitation phase. It starts exploiting the knowledge it has gained to achieve higher rewards. We do not want our agent to immediately go from one phase to the other however, because there might be some states which can only be accessed if the agent has made the "right" decisions. In our gridworld example, the states close to G can only be explored once the agent has learned to move safely past the D cells. Thus, we would like that our agents exploits its knowledge to avoid the D cells, but then still explores the states once it is past the danger.

There are different methods to balance this exploration-exploitation trade-off, the easiest one being the ϵ-greedy strategy. Here, there is a parameter ϵ which starts at a value of 1 and slowly decays over time to a certain (positive) small value. Every time the agent has to make a decision, it generates a random number uniformly between 0 and 1. If the random number is smaller than ϵ, it takes a random action (it explores), whereas if the number is larger is takes the action with the largest Q-value. As ϵ decays during the learning process, the agent will start by solely exploring the environment, but slowly over time it will favour exploiting its gained knowledge.

QL becomes quickly too computationally expensive. The gridworld has very limited number of states and actions, but as the number of states grow, the table becomes too large to hold in memory. This is especially true when dealing with continuous state representations, as these require a very rough discretization in order for the computer to be able to store the Q-table. Storage is not the only limitation however, as a large Q-table also implies a large learning-curve. Each entry has to be explored multiple times before we can start to trust the estimates. An approach that tackles both problems at once is Deep Q-learning. Instead of using a table, we can use a DNN to approximate the Q-values. This DNN is called the Deep Q Neural Network (DQN). Using a DQN we only need to learn the weights of the NN, which is much easier than learning all the Q-values themselves.

There are a lot of subtilities with DQN. Different variants of the algorithm exist, each of them introducing some new features to improve the algorithm. They are beyond the scope of this work, but the interested reader can find out more about them in [10].

2.4 Proximal policy optimization

In this project we will exclusively use proximal policy optimization (PPO) [11], which we briefly cover in this section. In this framework, we parametrize the policy π_θ with a vector θ. The weights are updated by the agent in order to maximise its objective, which in PPO is defined as

$$L_t(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right], \tag{2.13}$$

where S is an entropy bonus which ensures sufficient exploration and L_t^{VF} is a squared-error loss. The factors c_1 and c_2 are fixed coefficients. The first term inside the expectation value is

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right]. \tag{2.14}$$

Here epsilon is a hyperparameter that needs to be tuned and $\hat{A}_t$ is an estimator of the advantage function at time step t. The second term uses a clip function to remove the incentive to make very large policy updates, while the taking the minimum results in a pessimistic bound on the unclipped objective $r_t(\theta)\hat{A}_t$. The advantage function is defined as [12]

$$A(s,a) = Q(s,a) - V(s) \tag{2.15}$$

and tells us how well a certain action performs compared to the average value of the current state. Finally, we also have the probability ratio

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, \tag{2.16}$$

with θ_{old} the parameter vector of the policy before the update.

With this algorithm, the agent does not learn action-values, but instead it directly learns the optimal policy. This allows the agent to learn stochastic policies, which is very useful for our case. Note however that PPO is not the only algorithm that allows

for stochastic policies to be learned. The agent will be allowed to train with a certain number of interactions with the environment, after which we can test the trained model inside a standardised environment.

This standardised environment will compare the action a_t chosen by the agent to the optimal control action a_t^{opt}. These should be as close to each other as possible. Therefore, we opt for the reward function

$$R(a_t, t) = \exp\left(-|a_t - a_t^{\text{opt}}|\right). \tag{2.17}$$

To test a learned model, we run the model with this reward function for several episodes and record the total cumulative reward of each episode.

PPO was implemented in Stable-baselines [13], a library which contains an extensive range of RL algorithms created over the years. For this project, we start from the code created by F. Pino [4]. The environment is discretized in both spatial and time directions and was put into a class that can work with the Stable-baselines functions. We let the agent learn a policy by letting it interact with the environment a large number of times, periodically saving the best model it has learned so far.

2.5 Reward shaping

As stated in the reward hypothesis, we describe our intended goal or purpose by the maximization of the total cumulative reward. This means, that we must put good thought in how we design the reward function. If we reward certain actions, those actions will be repeated. Conversely, if we penalize certain actions the agent will not choose them as often. The tricky thing here, is that sometimes the agent finds ways to use certain actions in ways we as programmers had not anticipated. This happened in the example with the agent going from A to B and started to run around in circles. If we think carefully about the implementation of the reward function in that example, rewarding the agent every time it approaches B, running around in circles makes sense. If approaching B is your way of measuring progress, then circles are the optimal strategy. This shows that a good reward function is critical to arrive at the intended behaviour of the agent. We want to *shape* the reward in such a way that the agent experiences the problem we want to solve.

The reward shape is not only important to accurately describe to problem. As it turns out, it can massively impact the learning process itself. As of late, many RL researchers are trying to speed up learning by shaping their reward functions in new ways. An overview of the early research on this can be found in [14]. However, it was only with the introduction of Potential-based reward shaping (PBRS) in [15] that this topic became so important. The general idea is to introduce a second reward function F, which gives an additional reward on top of the one from R. In this new function, we can encode some prior knowledge about the problem and try to communicate this to the agent via rewards.

There are essentially two parts to reward shaping. First, we have some freedom to choose the function R as we see fit, as long as it correctly describes whatever it is what we want to achieve. Some things to consider here are first of all the complexity of the function. Built-in mathematical functions are easy to implement and fast to execute.

Surely we can write a very complicated function which captures a lot of fine details, but this function has to be evaluated in every step of the simulation. Making this reward function as easy as possible will help a lot in speeding up the algorithm, be it only from a computational standpoint. Next, we also have the choice to give either positive or negative rewards. Intuitively, one might guess that negative rewards are better, as they eliminate the risk of creating loops (as in the robot example from before). This will be one of the points we focus on in the project. Lastly, we can choose how we want to reward certain actions. Taking our robot as an example again, we can ask ourselves how we want to reward the agent if it takes a little step towards point B versus a bigger step. Has the reward to be bigger by a factor of 10? Or 100? This choice can also determine how fast the agent finds the optimal strategy, but may also restrict the agent in its exploration if tuned too harshly.

The second part is then to introduce a function F. There are many ways of doing this and finding the best one is still part of ongoing research. However, one thing is clear: it is very important to check that this function F does not change the objective. As we said before, the reward function describes the goal of the agent. By introducing F, we are changing the overall reward function and thus potentially the goal. When would it then be useful to introduce such a function? For example, if we are training an agent to play chess, it might take a long time to learn, because it only receives a reward at the end of each game. So it has to evaluate a whole bunch of moves using only 1 piece of information. But, we could give additional reward during the game. We could give a bonus for taking the opponents queen or by putting the opponents king in check. While this seems harmless, it might hinder the agents creativity. These bonuses push the agent towards a certain behaviour, that we think is the most optimal one. But human prejudice is quite often wrong.

2.6 Curiosity-driven learning

Curiosity-driven reward shaping [16] is inspired by human behaviour, which does not always need extrinsic rewards to explore or to learn. In psychology, we make the distinction between intrinsic and extrinsic motivation [17]. Both types drive us to do (new) things, but their origin is quite different. There is also a substantial difference in the effects of rewards in both cases. While this is most likely not generalizable to virtual agents, the concept of intrinsic motivation could prove useful in the evolution of RL algorithms [18]. Intrinsic rewards are especially helpful in environments with sparse rewards, as the lack of constant feedback makes it very hard for an agent to learn. By having a reward signal which does not depend on the environment, it can still assess its own actions during training.

In the curiosity-driven approach, an agent is pushed towards exploration, by rewarding it if it encounters states which it did not anticipated very well. In a sense, we want the agent to learn skills and gain new knowledge that might be handy later on. Knowledge about a certain state can be generalized by the agent. For example, two states which are very similar will probably have a similar optimal action. The method itself involves 2 NN, as shown in Figure 2.4. One NN, the *target* network, tries to embed the observation into a feature vector $\phi \in \mathbb{R}^n$ and predict the next action. By using the actual action chosen by the agent as the label, the target network uses this prediction to

update its weights. At the same time a second *prediction* network is used to predict the feature vector of the next state. A large difference between the predicted and the actual observed feature vector of the next state, indicates the agent has come across something it has very little knowledge about. This is exactly what the agent is supposed to do in this approach, so the (absolute value of) difference provides immediately the intrinsic reward

$$r_t^i = \frac{\eta}{2}\left\|\hat{\phi}(s_{t+1}) - \phi(s_{t+1})\right\|_2^2, \tag{2.18}$$

where $\eta > 0$ is a scaling factor and $\phi(s)$ indicates the feature vector of the state s. In some environments, especially ones involving video input, not all data from a state is relevant. To make sure the irrelevant data does not interfere with the learning process, it is removed and only the useful parts are kept in the feature vector for learning. This is task of the target network. The total reward provided at each step to the agent is the sum of the intrinsic and the extrinsic reward

$$r_t = r_t^i + r_t^e, \tag{2.19}$$

with the extrinsic reward r_t^e given by the reward function R.

The approach in this work was based on random network distillation [19], which does not train the target NN and hence, does not predict the action. Instead, the target network is initialized with random weights and kept fixed. Only the prediction network is trained to mimic the output of the target network. While this might lead to unwanted behaviour, as the feature extraction is purely random and thus important features might not be represented correctly, the authors of [19] found that standard gradient-descent was not able to mimic the target network so closely that this would be an issue. This approach is not part of the standard Stable-baselines library, but was developed by Matthias Konitzny and can be found on his Github [20].

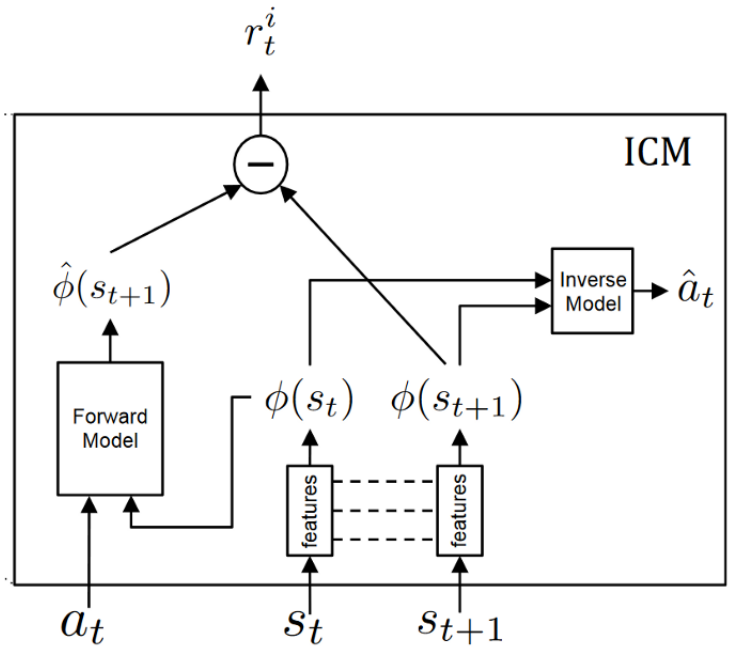

Figure 2.4: Schematic overview of the intrinsic curiosity module (ICM) from curiosity-driven learning. The prediction network from the text is marked "forward model" and the target network is considered to be the "inverse model" together with the feautre extraction. Figure taken from [16].

Chapter 3

Advection environment

Our goal was to train an agent for active flow control in a simplified 1D environment, inspired by the research done in [4]. We consider a spatial domain with a length of 50 m, with at $x = 5$ m a force creating a harmonic perturbation. The wave travels along the x-axis towards the agent, which controls an actuator located at $x = 18.2$ m and has to try to cancel the perturbation. For this task, the agent has access to 6 observation points. An overview of the setup is given in Figure 3.1.

The disturbance is advected inside the medium, according to the advection equation

$$\frac{\partial u}{\partial t} + a\frac{\partial u}{\partial x} = f(x,t) + c(x,t), \tag{3.1}$$

where u is the quantity advected by a, f is the force perturbing the medium and c is the control action. Advection is the transport of a quantity by bulk motion, so that the various properties are carried along with it. There are thus some conserved properties, such as the displacement and total energy. These conservation laws can be used to construct some unique reward functions, as we will see in the next Sections. The advection equation is an excellent playground to test and refine RL algorithms on, because it is one of the few control problems which can be solved exactly for a simple harmonic perturbation. We know exactly what the control action should be for a harmonic disturbance, namely the same harmonic but shifted with half a period. As a result, we can compare the actions of the trained agent to the true optimal control action and use that as a measure to compare performance among agents.

In order to apply RL to the problem, we must introduce a suitable reward function. The objective of the agent is to cancel the incoming wave. We can measure its performance by looking at the disturbance after the controller. We take the L^2 norm over a subset of the simulated spatial domain as the basis for different reward functions. We want of course the norm to be as close to 0 as possible. One option would be to simply use the opposite of the norm as the reward. In the following Sections we will investigate this and compare the performance of this choice against other possibilities for reward functions.

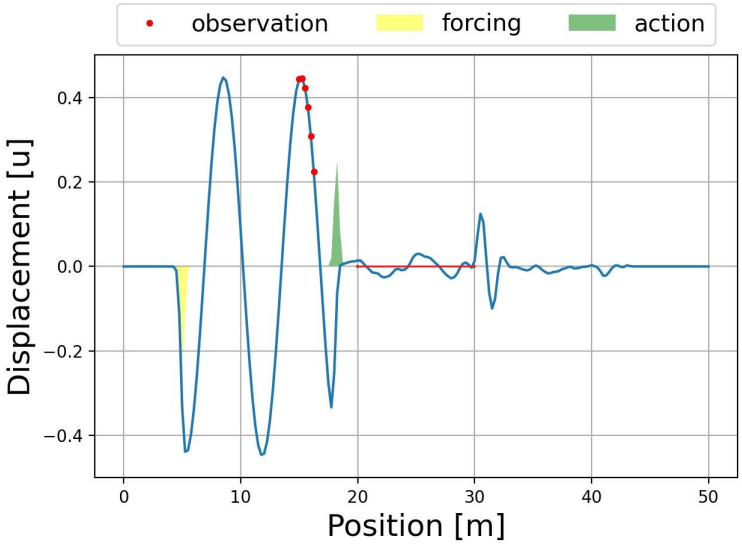

Figure 3.1: Setup of the advection equation environment. The agent controls the green controller and has access to the observations at the red dots. The L^2 norm is taken over the interval indicated by the red line.

3.1 Finding the optimal reward shape

The first step was to find a suitable reward function. The observable we used was the L^2 norm over the spatial domain $[20, 30]$m. The first idea was to simply use the opposite of the norm as the reward. We want the norm to be as close to 0 as possible, so using the aforementioned reward function, the agent receives a more negative reward the larger the norm is. As it is trying to maximize the reward, it will try to minimize the norm.

It is however not guaranteed that this reward function will lead to a fast convergence of the agent, due to the linear relationship between the norm and the reward. If the punishment of bad behaviour is not severe enough compared to the reward for good behaviour, the agent will not quickly find the good one. The incentive to drop a bad policy must be high enough in PPO, because an agent is discouraged to make drastic changes to its policy. This is necessary to avoid large policy updates, which would lead the agent outside the region in parameter space where it has just sampled. This idea is not unique to PPO, but was originally put forth in [21].

In Table 3.1 we give an overview of the different reward functions we tested. With each reward function, we ran the agent for 10 million iterations. During the run, the agent was periodically evaluated and if it performed better than the previous evaluation, the weights of the policy were saved. As such, we ended up with the best models for every reward function, learned over a period of 10 million iterations. We then loaded the best models and tested them on a standardized test reward function, which compares the action chosen by the agent to the known, optimal control action. Each model was run 100 times and the total cumulative reward for each episode was saved. In Table 3.2 we report the mean cumulative reward of the different models, over 100 episodes. We clearly see that negative reward functions make for better models. This is to be

Table 3.1: Table of the different reward functions which were tested. Here, l is the L^2 norm over the interval $[20, 30]$m and u is the displacement vector.

Name	Function
Linear negative	$-l$
Linear positive	$-l + 5$
Negative exponential	$\exp(-l) - 1$
Positive exponential	$\exp(-l)$
Gaussian negative	$\exp\left(-\frac{l^2}{2\sigma^2}\right) - 1$
Gaussian positive	$\exp\left(-\frac{l^2}{2\sigma^2}\right)$
Hyperbolic	$\text{clip}\left(\frac{1}{l}, 1, 10^5\right)$
Displacement conservation	$1 - \left\lvert \frac{u[x=18.2\text{m}]}{u_{\max}[x=18.2\text{m}]} \right\rvert$

Table 3.2: Mean cumulative reward over 100 episodes for the best models learned over 10 million iterations, with the different reward functions.

Name	Mean cumulative reward
Linear negative	31.26 ± 3.93
Linear positive	1.54 ± 0.87
Negative exponential	32.96 ± 4.12
Positive exponential	4.27 ± 1.37
Gaussian negative	28.57 ± 3.50
Gaussian positive	2.84 ± 1.28
Hyperbolic	0.86 ± 0.75
Displacement conservation	3.00 ± 1.18

expected, as the agent has all reasons to drop the bad behaviour as quickly as possible. The problem with positive rewards is that the agent still stacks up reward when doing the wrong thing (this is analogous to our example of the agent going from A to B in Chapter 2).

The mean cumulative reward of the three negative reward functions are all within 1σ of each other. There seems no preferred function. We can however look at another metric: the number of iterations required to learn the model. One of the big challenges in computational fluid dynamics is the time required to simulate the fluid. While numerous improvements have been made, the simulation part remains the bottleneck in most research projects. While the simulation of the advection equation evolution is not computationally expensive and does not represent a major bottleneck, this will not be the case for other environments. It is thus very advantageous if we can design a reward shape that minimizes the number of simulation steps required in order for the agent to learn a good policy. We report the number of steps at which the best model was saved for the three negative reward functions in Table 3.3.

From this, we can see that the agent which used *Gaussian negative* as reward function needed the least amount of iterations to find its best policy. Therefore, in what follows we will exclusively work with this reward shape.

Table 3.3: Numbers of simulation steps required to learn the optimal policy for the reward functions which yielded the best models in Table 3.2.

Name	Number of steps (in millions)
Linear negative	~ 6.7
Negative exponential	~ 8.5
Gaussian negative	~ 5.3

3.2 Enforcing continuous control

We can try to add a reward shaping function F to the *Gaussian negative* function, to see if we can speed up the convergence. First, we tried adding a continuity penalty to enforce a continuous action. This makes sense for our case, as the optimal control policy is a smooth function. While it seems a statement of an obvious fact, there are cases where this lead to a performance boost [22] and reduced the time required to train quite considerably.

The continuity reward function compares the last action to the action taken before that and penalizes the agent if these are too far apart. The maximum separation at which two actions are considered too far apart is a hyperparameter that needs to be tuned depending on the discretization of the problem. The closer the points are in space, the smaller the difference in action should be. Another important hyperparameter is the relative weight of the penalty. In the code, the action was chosen from $[-1, 1]$, which was then translated to a forcing action. Comparing two actions results then in small differences, thus we must multiply the difference with a weight to make it able to substantially alter the original reward. However, if this weight is too high, the agent will be focused on learning some continuous control action, instead of the one we want, as it is trying to avoid the continuity penalty.

After testing different weights, we found a configuration which was able to learn a model in ~ 7.3 million steps, which achieved 20.33 ± 3.61 in our standard test environment. While this looks like a bad result, we suspect that this agent should be able to converge faster to a good policy. We tested this hypothesis by training an agent with only 6 million iterations, instead of the previous 10 million. The results of these tests can be found in Table 3.4. The agent working with the vanilla Gaussian negative had a worse performance at this stage, compared to the agent which was shaped with the continuity penalty. In Table 3.5 we also check that given enough interactions, both models converged to the same optimal policy.

3.3 Energy conservation as a belief

Another technique that can be used to speed up reinforcement learning, is belief reward shaping (BRS). In this framework, an agent is equipped with an internal critic, which judges the agent actions based on prior beliefs [23]. The critic can modify the rewards coming from the environment with its own reward function and thus steer the agent towards a behaviour the critic beliefs to be the right one. The interesting feature of this framework is that it allows for prior knowledge about a certain problem to be incorporated into the agents decision making, without the agent needing to learn for

Table 3.4: Mean cumulative reward over 100 episodes for the best models learned over 6 million iterations, with the different shapes applied to the Gaussian negative reward function.

Name	Mean cumulative reward
Gaussian negative	13.32 ± 2.53
with continuity penalty	23.09 ± 3.37
with belief reward	25.78 ± 3.63
with both	25.87 ± 3.48
Curiosity-driven	1.58 ± 0.86

Table 3.5: Mean cumulative reward over 100 episodes for the best models learned over 10 million iterations, with the different shapes applied to the Gaussian negative reward function.

Name	Mean cumulative reward
Gaussian negative	27.01 ± 3.99
with continuity penalty	20.33 ± 3.61
with belief reward	29.65 ± 4.34
with both	32.96 ± 4.58

itself. However, the implementation is crucial, in order to not bias the agent too much to human prejudices.

The authors of [23] allow for the critic to update its beliefs as data is flowing in. We use a simplified framework, in which the critic modifies the environment reward based on a fixed belief related to energy conservation. As the energy travels linearly in the advection environment, the energy inside the wave at the observation points must entirely be counteracted by the controller, if we want to cancel the perturbation. In case of a harmonic disturbance, this means that the ideal control action is the same wave, but shifted with half a period. This amounts to a reflection of the wave around the x-axis. The wave of the control action must then match in incoming one, modulus a scalar. This belief can be incorporated by storing the past observations and comparing their evolution in time to the actions taken by the agent. The critic then penalizes the agent if the waves do not look similar.

Using the internal critic, the agent managed to learn its best model in ~ 6.7 million steps and achieved a mean score of 29.65 ± 4.34 in the test environment. While this is a bit longer than the previous reported ~ 5.3 million steps for vanilla Gaussian negative, the score is slightly higher, although the scores are within 1σ of each other. Nonetheless, the results are quite different when the agent is given only 6 million interactions to train on. The best model learned by vanilla Gaussian negative then only achieves a mean score of 13.32 ± 2.53, compared to the mean score of 25.78 ± 3.63 achieved by the Gaussian negative function with the belief reward shape we presented above.

We also trained an agent equipped with both the continuity penalty and the belief reward shape. This agent achieved a similar performance to both agents which had only one of the two enhancements, but seems to be limited by the worst performing reward shape. Nonetheless, more investigation is required on this topic. In the case of an advection governed environment, we do not observe any improvement combining the

two reward shaping functions.

3.4 Curiosity-driven approach

Lastly, we also implemented a curiosity-driven approach, which was detailed in Section 2.6. We used an agent equipped with vanilla Gaussian negative as the reward function, which played the role of external reward in the curiosity-driven framework. This agent was trained on 6 million iterations, but required a much longer real-world simulation time to do so, compared to the previous reward shaping methods. After the 6 million iterations, the best model achieved a score of 1.58 ± 0.86.

Looking at the entropy loss, which is a measure of how much the agent is exploring, we see that the agent is exploring until the very end. This might be the reason why the learned model performs so poorly, because it had not enough time to settle down on a good policy. This seems however to be a major downside to curiosity-driven learning. In other environments, where the computational cost of simulation can be much higher, we need an agent to find the optimal control action with the least amount of iterations. While curiosity-driven learning might very well-suited for problems with sparse rewards, it does not seem that it performs well in our case. We must note however that we used a specific implementation of curiosity-driven learning, namely RND. Other methods might perform better for our problem.

Chapter 4

Conclusion

4.1 The use of RL for active flow control

We applied a harmonic perturbation inside an advection governed environment in 1 dimension and trained an agent to cancel this perturbation. The agent was trained on 10 million iterations, using PPO and we tested different reward functions. We found that the Gaussian negative reward function resulted in a policy which was close to the known, optimal policy. Moreover, the policy was found in less than 6 million iterations, which was considerably faster than the agents equipped with other reward functions.

We tried to speed up the convergence towards the optimal policy of the agent by applying reward shaping. First we tried to apply a continuity penalty to the agent, in order to steer it towards a continuous control action. The agent equipped with this reward shape achieved a higher score than the one without the reward shaping, when given only 6 million iterations to train on. Next, we implemented a form of BRS, which used energy conservation as a prior belief. The internal critic compared the actions taken by the agent to the incoming wave and penalized the agent if the evolution in time was not similar. Using BRS, the agent managed to find a model which achieved a higher score on average, although the difference with the vanilla Gaussian negative was not significant. Again, the convergence seemed to be faster, as it achieves a higher score after only 6 million iterations. Lastly, we also implemented an RND based curiosity-driven approach. The results from our tests where not very promising. The agent required not only more simulation time, but also performed poorly in out standardised test environment.

More research is required to fully sample the potential of reinforcement learning for active flow control. One obvious part which could be further developed, is the implementation of the agents in other environments. The advection environment with a simple harmonic perturbation is a simple case. To go to more complex phenomena, one could try to use a perturbation which is a sum of different harmonics. Or instead of using advection, an environment governed by other equations could be used. Furthermore, the BRS function could be extended to allow for belief updates during training. This would probably require the use of a NN to represent the belief, such that it can be easily updated based on incoming data. Lastly, other reward shaping methods should be considered. An example could be to use another approach to curiosity-driven learning, as RND is only one of different existing methods.

References

[1] Edmond Wong. *Active Flow Control Laboratory*. 2008. URL: https://www.grc.nasa.gov/WWW/cdtb/facilities/flowcontrollab.html.

[2] 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 (Feb. 2019), pp. 281–302. ISSN: 1469-7645. DOI: 10.1017/jfm.2019.62. URL: http://dx.doi.org/10.1017/jfm.2019.62.

[3] R.Evans et al. *De novo structure prediction with deep-learning based scoring*. 2018. URL: https://deepmind.com/blog/article/alphafold-casp13.

[4] F. Pino, M. A. Mendez, and J. Rabault. *Combining Reinforcement Learning and Behaviour Cloning for Active Flow Control*. Apr. 2020. DOI: 10.13140/RG.2.2.24904.55043.

[5] Silver D. et al. "Mastering the game of Go with deep neural networks and tree search". In: *Nature* 529 (2016), pp. 484–489. DOI: https://doi.org/10.1038/nature16961.

[6] R. Tibshirani. *Gradient Descent: Convergence Analysis*. 2013. URL: http://www.stat.cmu.edu/~ryantibs/convexopt-F13/scribes/lec6.pdf.

[7] M. Stewart. *Introduction to Neural Networks*. 2019. URL: https://towardsdatascience.com/simple-introduction-to-neural-networks-ac1d7c3d7a2c.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016, p. 200.

[9] R. S. Sutton and A. G. Barto. *Reinforcement learning: an introduction*. MIT press, 2018.

[10] Thomas Simonini. *Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed...* 2018. URL: https://www.freecodecamp.org/news/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682/.

[11] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[12] G. Vadali. *Advantage function in Deep Reinforcement learning*. 2019. URL: https://medium.com/mindboard/advantage-function-in-deep-reinforcement-learning-39a0d809c1ff.

[13] A. Hill. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2020.

[14] Adam Daniel Laud. "Theory and application of reward shaping in reinforcement learning". PhD thesis. University of Illinois, 2004.

[15] Andrew Y. Ng, Daishi Harada, and Stuart Russell. "Policy invariance under reward transformations: Theory and application to reward shaping". In: *In Proceedings of the Sixteenth International Conference on Machine Learning*. Morgan Kaufmann, 1999, pp. 278–287.

[16] D. Pathak et al. "Curiosity-driven Exploration by Self-supervised Prediction". In: *Proceedings of the 34 th International Conference on Machine Learning*. Ed. by Mark Reid. 2017.

[17] Kendra Cherry. *Differences of Extrinsic and Intrinsic Motivation*. 2020. URL: https://www.verywellmind.com/differences-between-extrinsic-and-intrinsic-motivation-2795384.

[18] Zeyu Zheng et al. *What Can Learned Intrinsic Rewards Capture?* 2020. arXiv: 1912.05500 [cs.AI].

[19] Yuri Burda et al. "Exploration by Random Network Distillation". In: *CoRR* abs/1810.12894 (2018). arXiv: 1810.12894. URL: http://arxiv.org/abs/1810.12894.

[20] M. Konitzny (@NeoExtended). *Stable Baselines Fork*. https://github.com/NeoExtended/stable-baselines. 2020.

[21] John Schulman et al. *Trust Region Policy Optimization*. 2015. arXiv: 1502.05477. URL: http://arxiv.org/abs/1502.05477.

[22] A. Raffin. *Learning to Drive Smoothly in Minutes*. 2019. URL: https://towardsdatascience.com/learning-to-drive-smoothly-in-minutes-450a7cdb35f4.

[23] Ofir Marom and Benjamen Rosman. "Belief Reward Shaping in Reinforcement Learning". In: *In Proceedings of the Thirty-Second AAAI Conference on Artificial Interlligence (AAAI-18)*. 2018, pp. 3762–3769.