2D Continuous Control; Further Testing on Reward Machines

T.H.F. Stessen

Daily supervisor: Giovanni Varricchione
Second Reader: Natasha Alechina

Humanities Department
Utrecht University
The Netherlands
February 2023

Abstract

In this paper we will investigate how well the reinforcement learning technique Reward Machines (RM) introduced by Icarte et al. performs in a new, more complex, domain. When introduced RM's were tested on a continuous control domain that involved navigation in only one dimension. In the current paper, testing was done on a new continuous control environment, a variant of the Swimmer environment that involves an agent circling around a point in 2D space. Testing in this environment was done using Counterfactual Experiences for Reward Machines to speed up learning and showed how capable RM's are in environments that are often encountered in real-world domains such as robotics. Our results indicate that RM's using CRM perform very well in the Swimmer environment compared to the baseline algorithm. This shows that RM's are a viable approach in 2D continuous control environments.

Keywords: Reward Machines, CRM, Continuous Control, Reinforcement Learning

Contents

1 Introduction

The field of Reinforcement Learning (RL) has been able to tackle more and more problems (Sutton and Barto 2018). Initially the field focused on discrete finite domains using techniques such as value iteration and Q-learning. As time went on the field expanded into continuous domains. This meant tackling many real world problems such as self-driving cars (Ma et al. 2021) and controlling robotics (Andrychowicz et al. 2020) could be attempted.
One of the main challenges that RL still faces are domains with sparse rewards. Many reinforcement techniques require frequent feedback to learn efficiently. Especially as the domains get more complex agents struggle to get any reward at all. This happens because how well the agent can explore ways to get reward is dependent on how capable it already is. If the domain is too complex for reaching rewards by taking random actions then the agent will struggle to learn. This leads to a sort of 'catch-22' where the agent needs to explore to get reward and also needs rewards to be able to explore.
In the past various solutions to this problem have been found. Adjusting the reward function to provide more frequent rewards is an obvious approach, but this often requires specific knowledge about the domain. Usually all available knowledge is already used when constructing the reward function. Another approach introduces the concept of curiosity, where the agent gets a small reward for visiting new states. This approach also has multiple challenges, such as being drawn towards environments that appear new but that do not actually represent change (staring at a television that is showing white noise for example). Furthermore, it comes with the extra challenge of needing to predict which states are similar to each other. These challenges can be solved (Burda et al. 2018) but the solutions do not perform as well as other techniques on some environments.
In this paper, we will explore a different approach that uses Reward Machines (Toro Icarte et al. 2020). Essentially, they are finite state machines that represent the reward function. Reward Machines are based upon the intuition that when you work on a problem you often have some idea of what a solution looks like. Even if you do not precisely know what to do this idea can help you in deciding your next step. You might for example not know the fastest way to get to your new job but you know that finding your car keys will probably be part of your commute. Moreover, Reward Machines also make use of the idea that in the same situation you might want to do different things depending on your goal. For example, on the road leading to your job you might encounter a roundabout twice. Driving on the same roundabout you first take the east exit stop and grab some coffee, while the second time you take the north exit to actually head to work.
These intuitions translate smoothly to many RL domains. While the ways the agent can interact with the environment are often not fully understood, the reward function needs to be explicitly implemented by someone. This means that we could take our knowledge of the reward function and expose the agent

to it. Encoding this information as an RM and combining it with various techniques presented in (Toro Icarte et al. 2020) can allow the agent to exploit this knowledge to speed up learning. Additionally, encoding the reward function as an RM allows for complex reward functions to be represented in a standardised way. This can make it easier to design a reward function, knowing that the complexity of the scale of RM's can be encoded and learned from efficiently. These include non-Markovian reward functions that depend on more than just the current environment, but also on the history of the agent. Examples of such reward functions are often found in domains were order is important, such as in manufacturing or navigation.

Since Reward Machines can potentially help with the learning process of many RL different domains it also important to know what the limit of the technique is. Previously, testing was done on various domains, that include discrete domains, continuous domains with discrete actions and continuous control domains. Of these, the continuous control domains were the most complex. However, for continuous control testing was limited to domains that involved navigation in only 1 dimension, left and right. For many applications 1D is not sufficient. Especially in the domain of robotics, where you often have continuous control, navigation is often done in an environment that is at least 2D. Further testing is thus required to find out how well RM's can perform in these situations.

To do so, in this paper, we will apply the techniques introduced in (Toro Icarte et al. 2020) to a new domain that is navigationaly more complex. Firstly, we will describe how RM's work and how they can be applied to the domains that we are interested in. Specificaly, how we can combine it with DDPG to learn in a continuous control environment. Secondly, we will construct an environment that tests the limits of what RM's are capable of. For this we will use a variant of the Swimmer environment, a continuous control domain from the MuJoCo library. This domain has a 2D environment, allowing the agent to navigate in all radial directions. That provides the possibility of more complicated Reward Machines than the ones used in the continuous control domain that Icarte et al used. Finally, we will discuss some of the learning-rate increasing algorithms and explain why we will be using CRM for testing. Putting that all together, we will apply the DDPG algorithm with and without the use of CRM to a navigation task in the Swimmer environment. This will answer the question: How does the learning speed advantage of RM's hold up when both controlling the agent and navigating the environment are complex? We hypothesize that CRM will overcome the difficulty of navigating in 2D and will perform similarly to the previously tested domains.

Answering this question will provide new insight into the capabilities of RM's and will further the field of RL, an important part of Artificial Intelligence. With this new insight we provide additional knowledge of how certain learning tasks could or could not be tackled. This makes it so that designing certain RL systems, like a robot that picks up trash, is easier since more would be known about how suitable RM's would be for handling that task.

2 Preliminaries

In computer science, the sub-field of Reinforcement Learning (RL) focuses on agents that take actions in an environment to maximize their reward. These agents can learn from their experiences with interacting with the environment. They observe the reward they get when they take a certain action and from that they change the way they will act in the future. Ultimately this can result in an agent that always takes the optimal action, the one that maximizes the reward. This means that if we specify the environment, the agent and the reward we can train an agent to take close to optimal actions even if we do not know what those are ourselves.

Definition 1 *A Markov decision process (MDP) is a tuple* (S, A, T, R, γ) *with:*

1. S *a (finite) set of states, possibly with some labeled as terminal states*
2. A *a (finite) set of actions*
3. $T : S \times A \times S \to [0, 1]$*, a transition probability function*
4. $R : S \times A \times S \to \mathbb{R}$*, a reward function*
5. $\gamma \in [0, 1]$*, the discount factor*

Here $s \in S$ *is the current state,* $a \in A$ *an action the agent can take and* $s' \in S$ *the next state.*
Furthermore, an MDP satisfies the Markov assumption, meaning that the next state of the system is only dependent on the current state.

The environment that holds the agent can often be described as a Markov Decision Process, see definition 1. In such an environment an agent has a policy $\pi(s)$, which is possibly non-deterministic, that chooses an action for every state. When the agent then takes an action it transitions to the next state according to $T(s, a, s')$ and receives reward $R(s, a, s')$, where s is the current state, a the action and s' the next state.
As a whole the goal of RL is to make the agent learn an optimal policy π^*, a policy that, when started from any state and then followed, leads to the highest possible reward. To determine if a policy is optimal the Q-function can be used. The Q-function for a certain policy, $q^\pi(s, a)$, is defined as the expected discounted reward, $\sum_{s' \in S}[T(s, a, s')(R(s, a, s') + \gamma q^\pi(s', a')]$, when taking an action and then following the policy afterwards. Here the discount refers to the hyper parameter γ which discounts rewards the longer they take to reach. If $\gamma = 0$ the agent does not value future rewards and if $\gamma = 1$ then the agent evaluates future rewards the same as immediate rewards. Usually a value a little less than 1 is chosen, to make the agent look as far into the future as possible while still having it slightly prefer immediate rewards. It is known that

the Q-function associated with an optimal policy satisfies the Bellman equation: $q^*(s,a) = \sum_{s' \in S}[T(s,a,s')(R(s,a,s') + \gamma q^*(s',a'))]$ for every state and every action.

2.1 Reinforcement Learning Techniques

2.1.1 Value Iteration

If both $T(s,a,s')$ and $R(s,a,s')$ are known the problem of RL reduces down to value iteration. Value Iteration is a process where the expected discounted value is assigned to every state. For every state it looks at all possible actions and then chooses the action with the highest expected reward and assigns that value to the current state according to $V_{i+1}(s) = \max_a\{\sum_{s' \in S} T(s,a,s')(R(s,a,s') + \gamma V_i(s')\}$. Initially only the states with an non-zero reward, i.e. a certain reward, have a value but as the process goes on every other state gets assigned a value as well. This continues until the difference between V_t and V_{t+1} becomes smaller than some threshold, a pre-defined hyperparameter.

2.1.2 Tabular Q-Learning

In contrast with Value Iteration Q-learning is a method that does not require prior knowledge of $T(s,a,s')$ and $R(s,a,s')$. Instead Q-learning only relies on the agents experiences when interacting with the environment, it's thus 'model-free'. It initially chooses a random value for each $q(s,a)$. Then the agent starts interacting with the environment, on each timestep it takes an action s, receives a reward r and transitions to the next state s'. After that the q-value is updated according to: $q(s,a) = q(s,a) + \alpha(r + \gamma \max_{a'} q(s',a') - q(s,a))$. Here α is the learning rate, a hyperparameter. Usually this value starts out high and is decreased over time, as to allow it to learn as fast as possible without a single experience having an impact too large later on. The way the agent chooses its actions can have several different implementations, with ϵ-greedy being the most common. There the agent takes the action with the highest q-value, or a random action an ϵ amount of the time. Just like value iteration, Q-learning is guaranteed to reach an optimal policy, although only in when every state-action pair has been reached an infinite amount of times.

2.1.3 Deep Deterministic Policy Gradient (DDPG)

DDPG is a method that uses neural networks to be able to learn in continuous control environments (Lillicrap et al. 2015). This provides an advantage over tabular Q-learning which needs to track each state-action pair and is thus limited to smaller (finite) domains. When the state-space becomes continuous, such as when tracking position as a float, other approaches are needed. One notable approach is deep Q-learning, which can learn in a continuous environment and produces discrete actions. However, if not only the state space but also the action space is continuous then approaches such as DDPG need to be used.

DDPG uses the actor-critic framework, where the actor network is responsibly for learning the policy $\mu(s|\theta^\mu)$ and the critic network for the Q-values $Q(s, a|\theta^Q)$, where θ^i represents the learned parameters of network i. During training the agent acts according to $\mu(s|\theta^\mu)$ and in doing so generates experiences (s, a, s', r). These experiences are then used to train the critic network and in return the critic network is used to train the actor network.
Implementing this concept with the two networks directly connected to each other makes the learning process very unstable, for that reason DDPG uses two additional stabilisation techniques. Firstly, instead of directly giving the experience to the critic network it is first put in a so called experience replay buffer. This buffer holds a certain maximum amount of experiences, discarding the oldest one when it fills up. When the critic network is trained, it uses a random sample (or batch of samples) from the buffer. This stabilises the learning process by giving more uniform samples by breaking the temporal connection between them. Secondly, DDPG uses a two target networks, one for the critic and one for the actor. These are essentially copies of their respective counterparts that represent older versions of themselves. Their weights are only updated according to $\theta^{\mu'} = \theta^{\mu'} + \tau(\theta^\mu - \theta^{\mu'})$ and $\theta^{Q'} = \theta^{Q'} + \tau(\theta^Q - \theta^{Q'})$ with $\tau \in (0, 1)$ and usually with τ close to zero. DDPG uses these networks to update the weights of the critic network which slows down the impact of new experiences and stabilises learning.
Putting these techniques together the networks are updated as follows. The experiences are used to train the critic network by minimising the square error between $Q(a, s|\theta^Q)$ and $r + \gamma Q(s', \mu(s', |\theta^{\mu'})|\theta^{Q'})$ where $\theta^{\mu'}$ and $\theta^{Q'}$ are the weights of the respective target networks. In return the Q-value that the critic network gives for a state-action pair is used to train the actor network. The gradient for adjusting the weights is equal to the following:
$\mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}]$ where ρ^β is the state-visitation distribution.

2.2 Reward Machines (RM)

Using a type of finite state machine called a Reward Machine it is possible to expose the structure of a reward function to an agent (Toro Icarte et al. 2020). A reward function might consist of multiple steps that must be taken or requirements that must be met before the agent receives reward. At the time of designing the reward function these steps are always known as they must be programmed in explicitly. This applies to real world domains as well, as in the end the sensors that detect these steps output a digital signal (although what to do with uncertain sensors is still an area of research). Since the structure is known it is possible to redefine the reward function in a way that is favourable to the learning process of an agent.

Definition 2 *Given a set of propositions $\mathcal{P}$, a set environment states S and a set of actions A, a Reward Machine is a tuple $(U, u_0, F, \delta_u, \delta_r)$ with:*

1. *U a finite set of states*
2. *$u_0 \in U$ the initial state*
3. *F a finite set of terminal states, with $U \cap F = \emptyset$*
4. *$\delta_u : U \times 2^{\mathcal{P}} \to U \cup F$, a state transition function*
5. *$\delta_r : U \to [S \times A \times S \to \mathbb{R}]$, a state reward function*

A Reward Machine thus takes abstract descriptions of the environment in the form of propositions as input and gives a reward function as output. The inputs are the propositions in $\mathcal{P}$ that are true in the environment at that timestep, a truth assignment σ_t. They could be things like 'the robot has made coffee' or 'the robot has run out of battery'. The Reward Machine then transitions to the next state according to $\delta_u(u_t, \sigma_t)$. At every timestep the Reward Machine outputs a reward function according to $\delta_r(u_t)$ which might be a constant function but may also be a complex function that depends on the environment. When the robot delivers the coffee and the Reward Machine changes to reflect that, the reward function might be a simple '+100' but it could also be 'the temperature of the coffee in °C'. The Reward Machine keeps receiving inputs and outputting reward functions until the current state is an element of F. This means that never-ending tasks can also be implemented by defining $F = \emptyset$.
Since Reward Machines are structured as finite state machines they can describe reward functions up to the complexity of those that can be describes by regular expressions. Any Markovian reward function, one that is only based on the current (S, A, S) tuple, can still be described by a Reward Machine with just one state. However, more complex reward functions that depend on the state-action history $(S, A)^*$ can also be described by an RM. It can do this as long as it only needs to distinguish between histories that can be described by a finite set of regular expressions over the elements of $S \times A \times S$. This is because those regular expressions are part of a regular language, which are precisely those that can be accepted by deterministic finite state machines (Hopcroft, Motwani, and Ullman 2006).
These more complex reward functions allow RM to describe complex structures. These include loops, conditional statements and behavioral constraints, as can be seen in the task in section 3. For even more complex reward functions, such as counting the number of times a state has been reached, automata higher up the Chomsky hierarchy could be used. However it is not currently known how to apply techniques that exploit knowledge of these structures to speed up learning. Finally, to apply Reward Machines to an actual environment with an agent we still need one more thing, a labelling function $L : S \times A \times S \to 2^{\mathcal{P}}$. This function assigns truth values to each of the propositions in $\mathcal{P}$ after the agent took some action a and transitioned from s to s'. With this we can define an MDP with an RM.

Definition 3 *A Markov Decision Process with a Reward Machine (MDPRM) is a tuple* $(S, A, p, \gamma, \mathcal{P}, L, U, u_0, F, \delta_u, \delta_r)$ *with:*

1. S, A, p *and* γ *as defined in an MDP*
2. $\mathcal{P}$ *a set of propositional symbols*
3. $L : S \times A \times S \to 2^{\mathcal{P}}$, *a labelling function*
4. $U, u_0, F, \delta_u, \delta_r$ *as defined in an RM*

How an agent works with an MDPRM is very similar to how it works with an MDP. At every timestep the agent observes the environment while in state s_t and takes an action a_t. The agent then transitions to state s_{t+1} according to $T(s_t, a_t, s_{t+1})$ and receives a reward. The difference with an MDP is that the reward is not only based on (s_t, a_t, s_{t+1}) but also on the current state of the RM u_t, according to $R_t(s_t, a_t, s_{t+1})$, where $R_t = \delta_r(u_t)$. Finally the state of the RM is updated by evaluating what propositions hold in the environment according to $\delta_u(u_t, \sigma_t)$, where $\sigma_t = L(s_t, a_t, s_{t+1})$.
As an example of an MDPRM, we consider the office world gridworld presented in (Toro Icarte et al. 2020). This environment represents an office with an agent that can move in the four cardinal directions. In this environment two prerequisites are defined before the goal can be reached. The agent must collect both some coffee ☕ and the mail ✉ (in any order) before it can deliver them to the office, o. Furthermore, this environment also contains some vases $*$, which may not be broken. In this case $\mathcal{P} = \{☕, ✉, *, o\}$ and we forgo defining the patrol points from the original since we do not need them. The labeling function here is straightforward, $e \in \mathcal{P}$ is true iff the agent arrives at a location that is labeled as e in the environment.

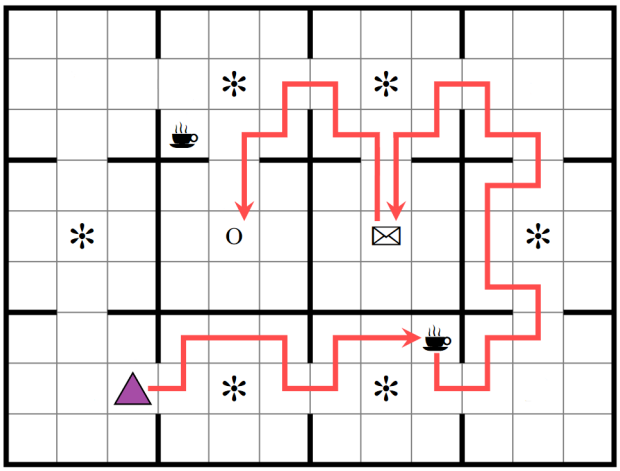

Figure 1: The Office Gridworld after Icarte et al. Shown here is the agent starting from a random position and a possible path it could take to complete the goal.

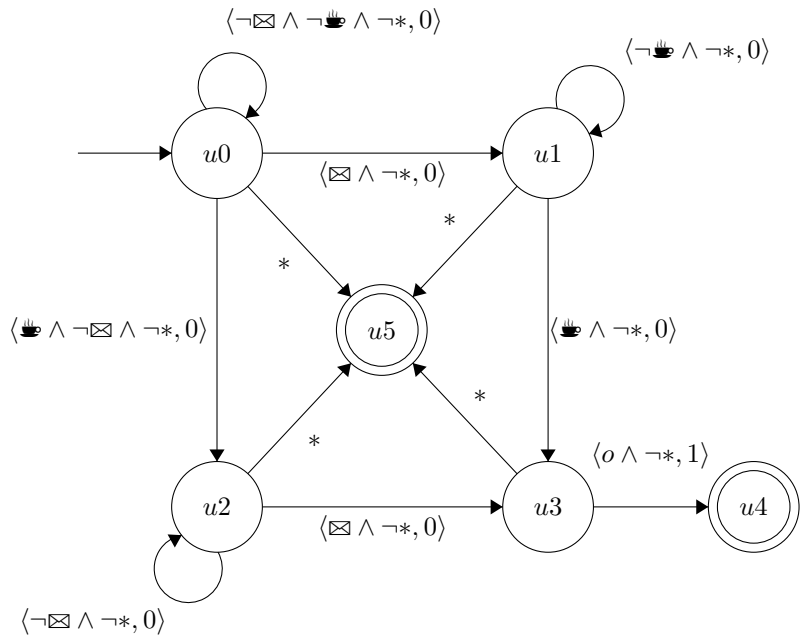

Figure 2: The corresponding Reward Machine for the Office World Environment. Here $* = \langle *, 0\rangle$, to increase readability. In each state, given a truth assignment, there is only one possible next state making this a deterministic finite state machine. Two terminal states are modeled, one is reached by breaking a vase (center), the other by completing the goal (bottom right).

2.2.1 Cross-Product Baseline (CPB)

For an agent to learn how to obtain the maximum reward in an MDPRM it needs to consider not only the current environment state but also the state of the Reward Machine. Thus the most straightforward approach would be to add the state of the RM to the inputs that the agent receives about the environment. Doing so allows any RL algorithm to learn a policy $\pi(\langle s, u\rangle)$, over the cross product $S \times U$. If the algorithm used is guaranteed to converge to an optimal policy for an MDP (such as Q-learning) then it is also guaranteed to do so for an MDPRM (See theorem 4.1 from Icarte et al.).
This technique thus makes it possible for any RL technique to make use of non-Markovian reward functions. For that, Reward Machines provide a neat standardised way to describe them. However CPB does not take advantage of the fact that the entire RM is known, so it does not provide a way to speed up learning.

2.2.2 Counterfactual Experiences for Reward Machines (CRM)

To take advantage of the knowledge encoded in the RM we will consider the technique Counterfactual Experiences for Reward Machines presented by Icarte et al. This technique also learns over the cross-product just as CPB. However the key difference is that CRM generates synthetic experiences for every state that the RM could be in. These experience can then be used by any off-policy RL algorithm, such as Q-learning and DDPG, to learn the policy $\pi(\langle s, u\rangle)$. Specifically, when an agent is in the state $\langle s, u\rangle$ and takes an action a and ends up in state $\langle s', u'\rangle$. Then for every state $u_c \in U$ we know that if the agent had been in state $\langle s, u_c\rangle$ then the agent would transition to state $\langle s', u'_c\rangle$, where $u'_c = \delta_u(u_c, L(s, a, s'))$. It would then receive reward $r_a = \delta_r(u_c)(s, a, s')$. Putting these together we can generate experiences for every RM state, instead of just the actual experience. Instead of giving $\langle s, u, a, r, s', u'\rangle$ to the agent we give it $\langle s, u_c, a, \delta(u_c)(s, a, s'), s', \delta_u(u_c, L(s, a, s'))\rangle$ for every $u_c \in U$.
Adding this process to an existing off-policy RL algorithm consists of adding just a few lines. See algorithm 1, an implementation for DDPG, where the RM state is added to the state and lines 11 to 14 are specifically for CRM. In general adding CRM means generating the experience and adding it to some sort of experience buffer.
The main driving force behind CRM's learning improvement is that CRM can explore the entire reward function while only exploring part of the environment. That way the agent already has experience with situations before it actually encounters them. Consider the example provided by Icarte et al. from the office world domain. There the robot reaches the office without yet acquiring coffee. Using just the CPB the agent would only learn that the office is not a good place to acquire coffee. An agent using CRM would instead also learn how to get to the office once it has met all the requirements.

Algorithm 1: DDPG with CRM

/* After (Lillicrap et al. 2015) */;
Randomly initialize critic network $Q(\langle s, u\rangle, a|\theta^Q)$ and actor network $\mu(\langle s, u\rangle|\theta^\mu)$ with weights θ^Q and θ^μ;
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$;
Initialize replay buffer R;
Initialize RM;
for *episode 1, M* **do**
    Initialise action exploration process $\mathcal{N}$;
    Receive initial state $\langle s_0, u_0\rangle$;
    for *t=1,T* **do**
        Select action $a_t = \mu(\langle s_t, u_t\rangle|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise;
        Execute action a_t and observe new state s_{t+1};
        for $u_{crm} \in U$ **do**
            Observe reward $r_t = r(s_t, a_t, s_{t+1})$ where $r_t = \delta_r(u_{crm})$;
            Store transition $(s_t, u_{crm}, a_t, r_t, s_{t+1}, u'_{crm})$ in R with $u'_{crm} = \delta_u(u_{crm}, L(s, a, s'))$;
        end
        Sample a random minibatch of N transitions (s_t, a_t, r_t, s_{t+1}) from R;
        /* as explained in 2.1.3 */;
        Update critic by minimizing the loss;
        Update the actor policy using the sampled policy gradient;
        Update the target networks;
        /* */;
        Calculate the truth assignment $\sigma_t \leftarrow L(s_t, a_t, s_{t+1})$;
        Update the RM $u_t \leftarrow \delta_u(u_t, \sigma_t)$;
        $s_t \leftarrow s_{t+1}$;
        if $u_t \in F$ **then**
            break;
        end
    end
end

3 Method

In this chapter we describe the environment and the techniques we used to evaluate the effectiveness of RM's. First we describe the environment that was used, the Swimmer environment, a continuous control domain from the MuJoCo library. Secondly, we describe what RL algorithms were used and how they were trained.
Together, the training process and the Swimmer environment provide a good way to test the boundaries of what RM's can do. This environment has both continuous observations and continuous control, making it difficult to learn in. Furthermore, previously testing on continuous control was limited to a domain where the agent could only move back and forth, effectively making the navigation 1D (Toro Icarte et al. 2020). The Swimmer environment provides the agent with a full 2D plane to navigate, thus putting it in untested territory. Finally, the comparison between the baseline, CPB, and CRM will show the difference in learning speed specifically due to CRM.

3.1 The Swimmer Environment

The Swimmer environment that was used consists of an agent swimming in circles around the origin. It is based upon the Swimmer environment from the OpenAI gym library (Brockman et al. 2016) and was originally introduced by (Coulom 2002). The current implementation runs in the MuJoCo simulator, a top of the line physics simulator.
This environment consists of a 2D plane that has a resistance as if the environment is covered in water. The agent always starts in the same location but with small perturbations in each run. The agent itself is a worm-like creature with two joints connecting three body parts of size 0.1. The end of one of its outer body parts is marked as its head and it receives additional information about that point. The agent can apply a real valued torque between -1 and 1 on each of its joints to wiggle through the 'water', hence the name swimmer. The inputs the agent receives are the same as in the default Swimmer environment, with the option for the (x,y) position turned on and with the RM state added. For a full overview of the information the agent receives see table 1 in the appendix.

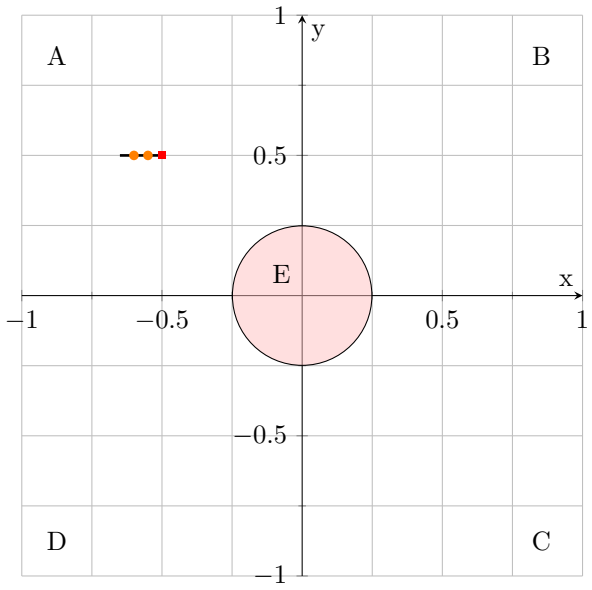

Figure 3: The Swimmer Environment. In the top left the agent is shown in its starting position before adding noise. The blue line represents its body, the two orange circles the joints it can move and the red square its head. The pink circle in the middle represents the 'kill zone' which, when the agents enters it with its head, ends the simulation and punishes the agent. The kill zone is labeled E and each quadrant is labeled A through D, corresponding with the labelling function.

To use this environment with an RM several additions were made, see figure 3. Firstly, each quadrant was labelled A through D. Secondly, a zone labelled E at $(0, 0)$ with diameter of 0.5 (bigger than the agent) was marked as a kill zone to avoid the agent spinning at the origin (see figure 6 in the appendix for results collected without the killzone). Accordingly, the labelling function assigns true to the corresponding letter for the area where the agent is located. Finally, the starting position of the agent is shifted to $(-0.5, 0.5)$, to place it outside the kill zone.
Finally the original reward function was replaced by a Reward Machine, see figure 4. This RM rewards the agent with 1000 points when it completes an entire lap. When the agent enters the kill zone it receives a punishment of 1000 points and the simulation ends. In any other transition the agent receives a small control penalty to discourage it from taking large actions, just as in the default implementation of the Swimmer environment. This penalty is calculated as follows $cp = 1e^{-4} * sum(action^2)$. If the agent never enters the kill zone the simulation runs for a maximum of 1000 timesteps. Note that there is otherwise no limit to the amount of points the agent can collect if it keeps completing laps.

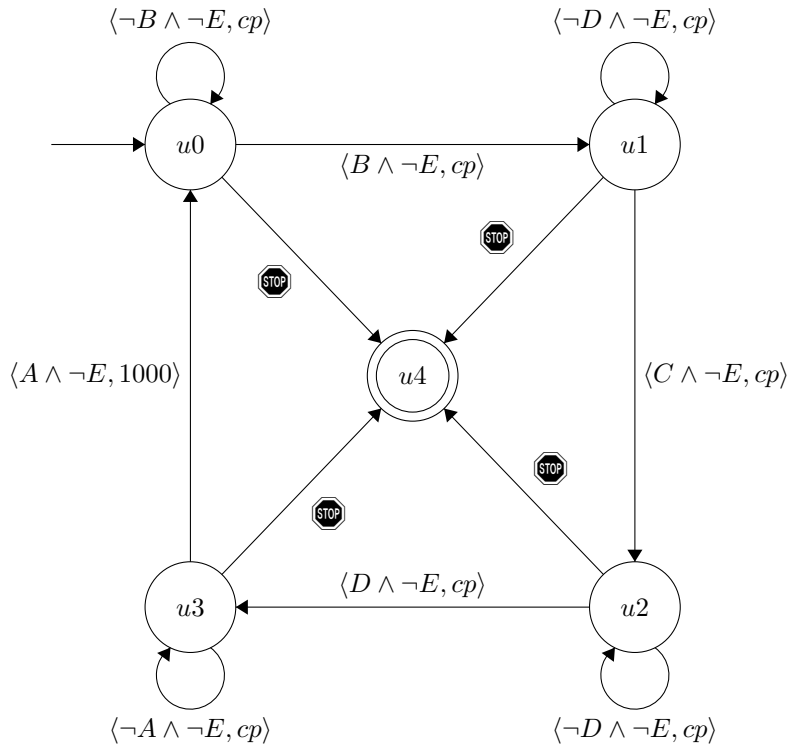

Figure 4: Reward Machine for the Swimmer environment. Here STOP = $\langle E, -1000 \rangle$ for readability. Here cp refers to a control penalty to discourage the agent from taking large actions.

3.2 Algorithms

In (Toro Icarte et al. 2020) various techniques were presented that take advantage of an RM to speed up learning. We will discuss their strengths and weaknesses and explain why we ultimately used CRM. Firstly, Hierachical Reinforcement Learning for Reward Machines (HRM) is a technique that decomposes the learning problem into subproblems which are simpler to solve. The advantage of HRM is that it can quickly learn to solve these subproblems resulting in a decent overall policy. This also comes with some disadvantages, namely that since it only looks at subproblems it can converge to a suboptimal policy. Moreover, the runtime of HRM is around 5 times higher than the other algorithms which puts the training time in order of days instead of hours. Secondly, CRM (as explained in section 2.2.2) performed very well in previous experimental testing, usually taking between a factor of 1 and 2 more training steps than HRM to receive consistent reward. Furthermore, training CRM does not take a significantly longer time to train than the baseline. However, previously in a specific task CRM did not perform well compared to HRM, although

this task was made to be very long and simple, which the Swimmer environment is not. Finally, Automated Reward Shaping (RS) is a technique that provides transforms the reward from the RM to be easier to learn from and it can be applied on top of another algorithm. This method was not used, simply because it did not improve the results for continuous domains and even lowered them in a few cases. Considering all these techniques, CRM was chosen to be the best to evaluate RM's in our new environment. It provides the best performance per hour trained and it was the best fit for the Swimmer environment.
Using the Swimmer environment CRM was compared to CPB using DDPG, from OpenAI baselines (Dhariwal et al. 2017), for both their underlying off-policy learning algorithm. Both algorithms were trained using the same settings. They used a feed-forward neural network 2 layers of 256 relu units, with a gamma of 0.99 and a batch size of 64. The rest of the hyperparameters were set to the default values from OpenAI baselines. The algorithms did 5 training runs of 3 million training steps, using the same 5 seeds for each algorithm. In accordance with the default parameters, these training steps were divided into 20 epochs, which each epoch running the agent through the environment 100 times.

4 Results

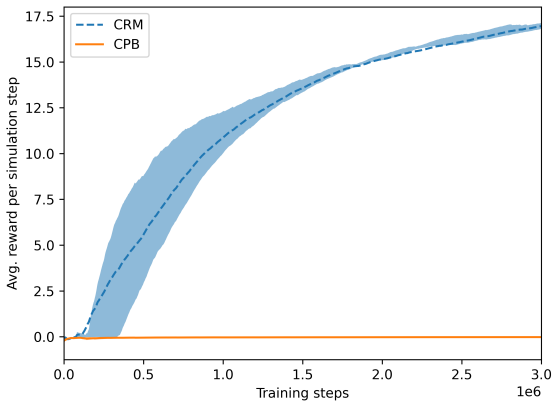

Figure 5: Results from testing on the Swimmer environment. On the x-axis the training steps are shown, while the y-axis shows the running average of the average reward the agent receives for every simulation step. The dashed blue line represents CRM and the solid orange line represents CPB. In both cases the line indicates the average over the 5 runs while the shaded areas indicate the 25th and 75th percentile quantiles.

The results indicate that CRM vastly outperforms CPB in the Swimmer domain, see figure 5. CRM quickly found a policy that could complete multiple laps, while CPB never achieved to learn a policy with consistent reward. Note that in some cases the average lies outside of the 25th and 75th percentiles for CRM. This is due to some runs vastly overperforming (around 100 thousand timesteps) or slightly underperforming (around 2.2 million timesteps).
To represent the results in the most similar way as was previously done in (Toro Icarte et al. 2020) we used the running average of the reward per simulation step for our graph. This is due to the fact that the github implementation by the authors of the Icarte et al. paper outputs two files: one containing the monitor and one containing simulation episode (rollout) information. The first of these does include the reward for each episode separately, but the amount of data points it outputs differs in relation with how many times the agent reaches a terminal state. Since this differs for each run, this makes it impossible to calculate the quantiles or the averages accurately without interpolation. The rollout file however does have a consistent amount of data points, but only records the running average. Since no interpolation was described in the Icarte et al. paper, we assumed they used the running average and thus we used it as well.

5 Discussion

The results show that CRM with DDPG can learn an effective policy in a continuous control domain with 2D navigation. This shows that RM's can perform well in environments that are more complex than previously tested.
In line with our hypothesis CRM learned a policy where the agent was able to complete multiple laps. This is similar to how CRM performed in the half-cheetah domain that (Toro Icarte et al. 2020) used for testing.
The fact that CRM performed so well in the Swimmer domain indicates that RM's are a suitable approach for various different domains. Specifically, in cases where there is a need for continuous control, 2D navigation and complicated rewards RM's can still perform well. These requirements arise in many real world domains such as robotics, which benefits from continuous control and often has intricate behavioral constraints.
While CRM performs well for the specific implementation of the Swimmer environment we acknowledge that it does not always do so. In cases where the scale of the environment is too large and the agent never reaches the first RM state transition the agent will struggle to learn. This kind of sparseness of reward RM's can not account for, however combining CRM with techniques such as curiosity could provide a fruitful path to a solution for this problem.
Since we have only found results that argue for the use of RM's we can not define what their upper bound, in terms of environment complexity, is. Further research could be done to investigate this more closely and to discover the limits of RM's. To give two examples, 3D environments and MDPRM's with many nodes could be tested. A 3D environment could be modeled using the Ant environment from the MuJoCo library. This would provide insight into the performance of RM's for the most complicated control tasks. Testing MDPRM's with a large amount of nodes could show how the runtime of the training process scales and if learning remains stable with many nodes.

References

[1] Rémi Coulom. "Reinforcement learning using neural networks, with applications to motor control". PhD thesis. Institut National Polytechnique de Grenoble-INPG, 2002.

[2] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. "Automata theory, languages, and computation". In: *International Edition* 24.2 (2006), pp. 171–183.

[3] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[4] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[5] Prafulla Dhariwal et al. *OpenAI Baselines*. `https://github.com/openai/baselines`. 2017.

[6] Yuri Burda et al. "Exploration by random network distillation". In: *arXiv preprint arXiv:1810.12894* (2018).

[7] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: `http://incompleteideas.net/book/the-book-2nd.html`.

[8] OpenAI: Marcin Andrychowicz et al. "Learning dexterous in-hand manipulation". In: *The International Journal of Robotics Research* 39.1 (2020), pp. 3–20.

[9] Rodrigo Toro Icarte et al. "Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning". In: *arXiv preprint arXiv:2010.03950* (2020).

[10] Xiaobai Ma et al. "Reinforcement learning for autonomous driving with latent state inference and spatial-temporal relationships". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE. 2021, pp. 6064–6071.

6 Appendix A

	Observation
1	Angle of the head
2	Angle of the first joint
3	Angle of the second joint
4	Velocity of the head along the x-axis
5	Velocity of the head along the y-axis
6	Angular velocity of the head
7	Angular velocity of the first joint
8	Angular velocity of the second joint
9	x position of the head
10	y position of the head
11	Current state of the Reward Machine

Table 1: Inputs that the agent receives in the Swimmer environment. Inputs 1-10 have a range of $(-\inf, \inf)$, while input 11 has a range of U. Observation 11 was added for compatibility with an RM.

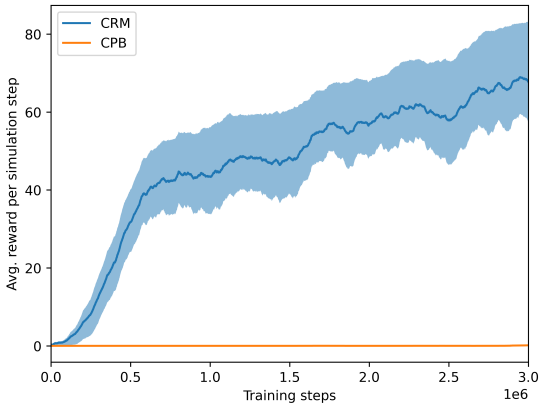

Figure 6: Results on the Swimmer environment with an accidentally disabled killzone. Each algorithm did 3 runs. Here, the y-axis represents the average reward per simulation step in contrast with figure 5, otherwise the same. Note how the reward is around 4 times higher than with a kill zone.