A Survey on Reinforcement Learning Methods in Character Animation

Ariel Kwiatkowski, Eduardo Alvarado, Vicky Kalogeiton, C. Karen Liu, Julien Pettré, Michiel van de Panne, Marie-Paule Cani

EUROGRAPHICS 2022
D. Meneveaux and G. Patanè
(Guest Editors)

Volume 41 (2022), Number 2
STAR – State of The Art Report

A Survey on Reinforcement Learning Methods in Character Animation

Ariel Kwiatkowski[1], Eduardo Alvarado[1], Vicky Kalogeiton[1], C. Karen Liu[2], Julien Pettré[3], Michiel van de Panne[4] Marie-Paule Cani[1]

[1] LIX, École Polytechnique/CNRS, Institut Polytechnique de Paris, Palaiseau, France
[2] Stanford University, Stanford, CA, USA
[3] Univ Rennes, Inria, CNRS, IRISA, Rennes, France
[4] University of British Columbia, Vancouver, Canada

Abstract

Reinforcement Learning is an area of Machine Learning focused on how agents can be trained to make sequential decisions, and achieve a particular goal within an arbitrary environment. While learning, they repeatedly take actions based on their observation of the environment, and receive appropriate rewards which define the objective. This experience is then used to progressively improve the policy controlling the agent's behavior, typically represented by a neural network. This trained module can then be reused for similar problems, which makes this approach promising for the animation of autonomous, yet reactive characters in simulators, video games or virtual reality environments. This paper surveys the modern Deep Reinforcement Learning methods and discusses their possible applications in Character Animation, from skeletal control of a single, physically-based character to navigation controllers for individual agents and virtual crowds. It also describes the practical side of training DRL systems, comparing the different frameworks available to build such agents.

CCS Concepts

• ***Computing methodologies*** → ***Reinforcement learning; Animation;***

1. Introduction

Computer Graphics (CG) and Virtual Reality (VR) applications, from movies to video games, make a wide use of virtual characters, i.e. digital representations of humans, animals or other living creatures. For a long time, animation pipeline standards have pursued realism and control over motion style through fully kinematic characters, often designed manually by artists specifically for each situation, resulting in high time and resource costs. However, the increasing complexity of many applications makes the development of more versatile authoring tools a priority. In particular, simulators, games and VR environments share the need for autonomous characters, able to act in the expected way, while being reactive to any changes in their environment due to the user's actions. In order to produce such systems, learning-based approaches need to be explored.

Modern Machine Learning is commonly divided into three categories: Supervised Learning (SL), Unsupervised Learning (UL), and Reinforcement Learning (RL). Supervised Learning refers to learning using data with labels, Unsupervised Learning, including Self-Supervised Learning makes use of raw data without labels, and Reinforcement Learning does not use data in the usual sense. Instead, the learning stage in RL consists of an agent taking a sequence of actions in one or more environments, and trying to maximize a reward function dependent on the states it visits. During this process, the agent progressively trains its own controller module, which in the case of Deep Reinforcement Learning (DRL) is represented by a deep neural network. Once learned, the network can be used in a new, and possibly evolving environment, to make the agent take actions in a successful way towards its goals.

RL stands out as a promising approach for character animation because it provides a versatile framework to learn motor skills without the need of labelled data. RL is particularly useful when the dynamic equations of the environment are unknown or non-differentiable, to which conventional gradient-based optimal control algorithms do not apply.

Compared to traditional methods in AI, the designer does not need to specify what the character should do in each case – a time-consuming and non scalable method. In contrast, the agent will discover the appropriate actions during the learning stage, given the targeted task or goals expressed in the form of a reward function.

This survey reviews the most common modern **DRL** algorithms, and how they can be used to tackle the main challenges in **character animation**. We consider two main categories of tasks – individual motion skills, and motion planning tasks. Individual scenarios typically involve skeletal motion control of a physically-based character, while motion planning often involves multiple characters

interacting in a shared environment. In particular, we focus also on the problem of **crowd simulation**, which focuses on determining the trajectories of multiple agents in a shared environment, often abstracting away their internal structure.

We begin by describing the main, most recent challenges in the field of character animation (Sec. 1.1). Then, we present the key principles and notations in RL (Sec. 2), and continue with a general classification of the most common approaches (Sec. 3). Subsequently, we divide the addressed RL solutions into two groups: single-agent (Sec. 4) and multi-agent (Sec. 5) problems. Then, we describe how these methods are used to solve computer animation problems, for skeletal motion control (Sec. 6) and navigation problems (Sec. 7), as well as some works concerning interactions between virtual agents and humans (Sec. 8). Finally, we present a description of current, available frameworks to apply RL-based solutions (Sec. 9), before concluding with a summary of the most relevant algorithms used for a particular problem.

Our work is largely complementary to a recent survey on deep learning for skeleton-based human animation [MHC*21], which we also recommend to readers. In particular, we provide a detailed review of current RL methods (both single agent and multiagent) and their mathematical foundations, a full review of RL methods for character navigation methods, and a complementary classification of physics-based character RL methods.

1.1. Problems in Character Animation

In the most general sense, the field of **Character Animation** concerns everything related to animating virtual characters. In this work specifically, we focus on the aspects of **behavior** of said agents, on their skeletal motion control, as well as on their **interactions** with a possible human user. Topics related to modeling and animating the character's face, skin, muscles, hair and clothes, or rendering it are out of scope of this report.

When dealing with a single animated character (which may also encompass situations with several independent characters), there are two main levels that need to be considered:

- Skeletal Animation
- Character Motion Planning

Skeletal Animation deals with internal motions of an agent – how the individual limbs move, while the position in the global frame may be of secondary concern. Character Motion is the opposite – it abstracts away the details of the character's shape, instead focusing on its displacements through the scene.

When considering Character Motion for multiple interacting characters, the problem turns into that of Crowd Simulation. Typically, in those problems, each agent has a destination it wants to reach, while avoiding collisions with the environment and with other agents. Van Toll and Pettré [TP21] wrote an overview of the modern approaches from the last decade.

2. Definitions and Preliminaries

In this section, we introduce the basic formal background of Reinforcement Learning. First, we describe and compare different ways of formalizing the RL task to specify **what** we want to solve. Then, we describe the fundamental theorems supporting modern RL methods to show **how** we can solve those tasks.

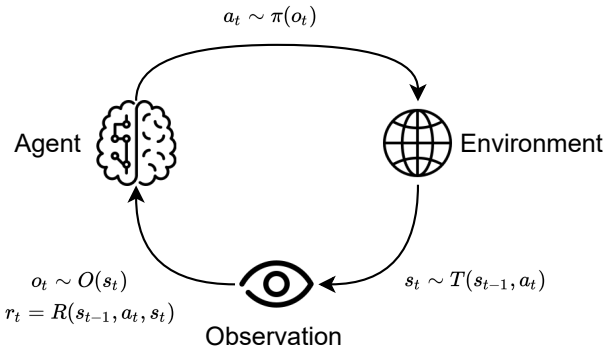

Figure 1: *A visual depiction of the basic Reinforcement Learning loop corresponding to the POMDP formalism. The agent and the environment exchange information between each other. The agent perceives the environment state and executes an action. The environment then updates its state, and communicates it to the agent via an observation function, together with the reward for the last action.*

2.1. Reinforcement Learning Formalisms

While there exist several frameworks that are used to formalize the Reinforcement Learning problem, they are based on the Markov Decision Process [Bel57, SB98, SB18] (MDP), with variations adapting it to the specific task at hand. In this section, we describe the variants relevant to character animation, both for individual agents, as well as multiagent scenarios.

In essence, a Reinforcement Learning problem consists of two parts – an **environment**, and an **agent** acting within that environment in order to achieve some goals. The agent observes the environment, receiving its **state** or **observation**, and based on that executes an **action**. The state of the environment then changes, and the agent receives a reward signal indicating how good that action was. The agent's objective is maximizing the total reward collected during an episode. An episode starts from an initial state, and lasts until the agent reaches a terminal state, or the environment terminates otherwise (e.g. due to a time limit). A schematic representation of this loop is in Figure 1.

2.1.1. Single Agent

A general **Markov Decision Process** (**MDP**) is defined by a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, \mu)$, optionally with a sixth component γ (which can also appear in all other formalisms, and hence will be omitted from their descriptions), where:

- $\mathcal{S}$ is a set of states of the environment.
- $\mathcal{A}$ is a set of actions available to the agent.
- $T\colon \mathcal{S} \times \mathcal{A} \to \Delta\mathcal{S}$ is the environment transition function, representing its dynamics.
- $R\colon \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function which is used to define the agent's task.

- $\mu \in \Delta\mathcal{S}$ is the initial state distribution.
- $\gamma \in [0,1]$ is the discount factor.

Note that we use the notation ΔX to represent for the set of all probability distributions over the set X.

During an episode, an initial state $s_0 \in \mathcal{S}$ is sampled from μ. The state is typically represented by a continuous vector in $\mathbb{R}^n$, or in simple cases, a discrete value. After which the agent repeatedly selects an action a_t from $\mathcal{A}$, observes a new state $s_{t+1} \sim T(s_t, a_t)$ and receives a reward $r_t = R(s_t, a_t, s_{t+1})$. The actions, similarly to observations, are typically continuous vectors or discrete values, although more complex nested structures are also used. This can repeat infinitely, or until some termination condition, defined either by a terminal state in $\mathcal{S}$, or a time limit. The agent's objective is maximizing the **total discounted reward** $\sum_t \gamma^t r_t$, or simply non-discounted **total reward** $\sum_t r_t$ when $\gamma = 1$.

The solution to an MDP is defined as an **optimal policy**, typically denoted as $\pi^* : \mathcal{S} \to \Delta\mathcal{A}$. It is the policy that, when executed, leads to the highest expected total discounted reward. While a policy may be stochastic or deterministic, depending on the properties of the action distributions it outputs, note that the optimal policy is generally stochastic, i.e. it returns a distribution over actions rather than a specific action. For consistency, the notation we use in this work is that the action distribution of a policy π in a given state s is $\pi(s)$, whether that policy is stochastic or not. The action is then sampled from the policy $a \sim \pi(s)$. An alternative notation uses the notion of a conditional probability of the action given the current state $\pi(a|s)$, and is equivalent to ours.

A key property of MDPs is their full observability - agents have complete information of the current environment state. This is rarely the case in real applications, and thus a **Partially Observable Markov Decision Process** [KLC98] (**POMDP**) is often used instead.

A POMDP is defined by a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, \Omega, O, \mu)$, where $\mathcal{S}, \mathcal{A}, T, R, \mu$ are defined as in MDPs. Ω is a set of possible observations, and $O : S \to \Delta\Omega$ is the observation function mapping states to observations. This time, the agent does not perceive the real state s_t of the environment, but rather the observation $o_t \sim O(s_t)$ which may not contain the full information, hence the partial observability.

2.1.2. Multiagent

While the MDP and POMDP formalisms are sufficient for problems with a single agent, the generalization to multiple agents can be done in different ways depending on the extent of flexibility required for a given application. The most general case is a **Partially Observable Stochastic Game** [HBZ04] (**POSG**) which is defined as a tuple $\mathcal{M} = (\mathcal{I}, \mathcal{S}, \{\mathcal{A}_i\}, \{\Omega_i\}, \{O_i\}, T, \{R_i\}, \mu)$, where:

- $\mathcal{I}$ is the finite set of agents, indexed $1, \ldots, n$
- $\mathcal{S}$ is a set of states of the shared environment.
- $\mathcal{A}^i$ is a set of actions available to agent i, and $\mathcal{A} = \times_{i \in \mathcal{I}} \mathcal{A}^i$ is the joint action set.
- Ω^i is the set of observations available to agent i.
- $O^i : \mathcal{S} \to \Omega_i$ is the observation function for agent i.
- $T : \mathcal{S} \times \mathcal{A} \to \Delta\mathcal{S}$ is the environment transition function, representing its dynamics.
- $R^i : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function of agent i, which defines the agent's task.
- $\mu \in \Delta\mathcal{S}$ is the initial state distribution.

Similarly to the single-agent scenario, the environment is initialized with a state s_0 sampled from μ. Each agent then receives an observation $o^i_t = O_i(s_t)$ and based on that, chooses an action a^i_t. The environment changes according to the joint action of all agents $a_t = (a^1_t, a^2_t, \ldots, a^n_t)$ generating the new state s_{t+1}, and each of them then receives their rewards $r^i_t = R_i(s_t, a_t, s_{t+1})$ and observations o^i_{t+1}. This repeats until the episode ends. Since each agent receives its own reward, the objective of an agent i is maximizing its total reward $\sum_t r^i_t$.

A special case of POSG is a **Decentralized Markov Decision Process** [BZI00] (**DecPOMDP**) in which all agents work together to optimize a shared reward function $\forall_i R_i = R$. This formalism is suitable for fully cooperative tasks. It is worth noting that any POSG can be converted into a POMDP by setting the reward to be equal to the sum of of individual rewards, but it will not make sense in all POSGs (consider for example any zero-sum game).

An alternative, but equivalent to POSG formulation, is the **Agent Environment Cycle Game** [TGB*21] (**AEC**) formalism. As opposed to the previous options, it is more adapted to dealing with environments in which agents do not act simultaneously. Formally, an AEC is defined by a tuple $\mathcal{M} = (\mathcal{I}, \mathcal{S}, \{\mathcal{A}_i\}, \{\Omega_i\}, \{O_i\}, P, \{T_i\}, \{R_i\}, \nu, \mu)$, where $T_i : \mathcal{S} \times \mathcal{A}_i \to \mathcal{S}$ is a deterministic agent transition function, $P : \mathcal{S} \to \Delta\mathcal{S}$ is the environment transition function, $\nu : \mathcal{S} \times \mathcal{I} \times \mathcal{A} \to \Delta\mathcal{I}$ is the *next agent* function which determines which agent will be taking the action next. The other symbols are defined as before, with the exception of $\mathcal{I}$ which now also includes environment itself considered as a separate agent, represented by the symbol 0. Furthermore, $\mathcal{A}$ is now an union of all individual action spaces. All agents, including the environment, take turns taking their actions and modifying the shared state, which enables a greater flexibility compared to the POSG formalism. AEC environments are primarily used in the Petting Zoo framework [TBJ*21] (see Section 9).

In some cases, a more game theory-based approach is useful. The **Extensive Form Game** [LLL*20] (**EFG**) formalism is notably used in the OpenSpiel framework. It contains implementations of many board games, which is the context that it excels in. However, it is not very applicable to character animation, and thus we refer the reader to the associated paper for further details on this formalism.

Note: Many details of the described formalisms vary between sources in the ordering of their elements, the size of the tuple, and the signatures of the functions. This does not change the underlying behavior, and we will therefore omit discussing the different descriptions of the same formalism.

2.1.3. Environment design

A crucial element when applying RL to new problems is designing an appropriate environment. This often involves building a simulation that implements the common API of Gym (see Section 9.2), since a purely mathematical formulation would quickly become

very convoluted in a complex scenario. Note that we omit the transition function from this description, as this is typically part of the underlying simulation, and can therefore be implemented in any way.

The first consideration is the observation space. This is commonly represented as a fixed-size vector space $\mathbb{R}^n$, which can be directly used with regular feed-forward neural networks. More complex nested structures as well as images are also possible, but they require an adaptation in the structure of the policy being learned.

Second comes the action space. Depending on the environment, a common choice is either a vector space $\mathbb{R}^n$, or simply a finite set of actions $|\mathcal{A}| = n < \infty$. While from the point of view of the implementation it is important that the action space remains constant between different states, one can employ invalid action masking to restrict the available actions to a specific subset. Similarly to observations, it is also possible to use nested structures as long as the policy is adapted correspondingly.

Finally, the reward function defines the actual task and guides the agent's behavior. This is often the most critical component to develop, as a misspecified reward function can lead to unexpected and undesirable behaviors. The simplest reward function can be obtained by choosing a goal state, and giving the agent a reward of 1 if it reaches that state, or 0 otherwise. However, this sparse reward tends to make it very difficult for the agent to learn, as it needs to reach it with random exploration to receive any training signal. A common method is then using reward shaping [NHR99] by adding a smaller, dense reward that guides the agent towards the goal. In other cases, there might be a natural dense reward that can be used instead of the sparse one, such as the distance from the goal in environments with relatively simple dynamics.

2.1.4. Summary

We presented the most commonly used formalisms underlying the RL problem, which serve as a basis for finding ways to solve these tasks. The similarities and differences between them are in Table 1. Typically, either MDP or POMDP can be used with a single agent. POMDP offers stronger theoretical justification if the agent does not observe the full environment state, but this high rigor is not always necessary. Instead, MDP is often used due to its simplicity. With multiple agents, POSG is a versatile choice that can work with any scenario. If one needs to put an emphasis on some aspect of the environment, other options are also available. It is worth noting that those formalisms have dynamic programming solutions associated with them for cases with discrete action and state spaces. This however is impractical in complex scenarios that emerge in character animation, requiring more sophisticated algorithms.

2.2. Fundamentals of RL Algorithms

In this section we describe the mathematical theorems underlying the most important RL algorithms used today. Specifically, we show how the **Policy Gradient Theorem** enables directly optimizing a behavior policy function, and the **Bellman Equation** enables learning the expected utilities of actions that the agent can take in a certain state. These will serve as a basis for many modern algorithms, which often combine the two aspects. We use the notation of MDPs described in Section 2.1.1 because they provide sufficient generality. Under partial observability, states are replaced with observations, and multiagent extensions of relevant algorithms are discussed in Section 5.

In both cases, modern algorithms use Neural Networks as approximators for the relevant functions. Because a detailed explanation of training neural networks is out of the scope of this work, we refer the readers to e.g. the Deep Learning Book [GBC16] for more information on that topic.

2.2.1. Policy Gradients

The Policy Gradient Theorem is a basis for all **Policy Gradient** (**PG**) algorithms, starting with the seminal REINFORCE algorithm [SMSM99]. In the context of deep reinforcement learning, the policy $\pi\colon \mathcal{S} \to \Delta\mathcal{A}$ is represented as a neural network, and its free parameters, e.g., the weights, are optimized using gradient ascent on the total expected reward. In order to do that, we need to find the gradient with respect to the network's weights using a batch of collected experiences. Here we show a proof of the theorem based on that published in OpenAI Spinning Up [Ach18], although other approaches for proving the same result exist [Wil92, Jon20].

Consider a trajectory in the environment, defined as a sequence of consecutive states and actions taken by the agent, and rewards $\tau = (s_0, a_0, r_0, s_1, a_1, r_1 \dots)$. Given the parametrized policy π_θ, we know that the probability of a trajectory is

$$P(\tau) = \mu(s_0) \prod_t P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{1}$$

$$\log P(\tau) = \log\mu(s_0) + \sum_t \left(\log P(s_{t+1}|s_t, a_t) + \log\pi_\theta(a_t|s_t)\right) \tag{2}$$

and the total reward obtained in the trajectory is $R(\tau) = \sum_t r_t$

Consider now the expectation across all trajectories τ. With the optimization target defined as $J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta} R(\tau)$, using a few calculus transformations, we can express the policy gradient as:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} R(\tau) && (3)\\
&= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) && (4)\\
&= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau) && (5)\\
&= \int_\tau P(\tau|\theta) \nabla_\theta \log P(\tau|\theta) R(\tau) && (6)\\
&= \mathbb{E}\left[\nabla_\theta \log P(\tau|\theta) R(\tau)\right] && (7)\\
&= \mathop{\mathbb{E}}_{\tau\sim\pi_\theta} \left[\sum_t \log\pi_\theta(a_t|s_t) R(\tau)\right] && (8)
\end{aligned}$$

With this, given a batch of trajectories $\mathcal{D}$ collected using the policy we are optimizing, we can finally compute the gradient estimate:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau\in\mathcal{D}} \sum_t \nabla_\theta \log\pi_\theta(a_t|s_t) R(\tau) \tag{9}$$

Table 1: *A comparison of different formalisms used to define an RL problem. Legend: × – the property cannot be modelled in this formalism, ∼ – the property can be modelled in this formalism, but is not the intended use or requires extra effort, ✓ – the property can be modelled in this formalism, ★ – this formalism is particularly suitable for this property. Multiagent Cooperative and Competitive refers to the rewards being either shared or zero-sum, respectively. Multiagent Mixed is neither fully cooperative nor competitive. Simultaneous and Turn-based refers to whether all agents take their actions at the same time, or only one agent does.*

Property	MDP	POMDP	POSG	DecPOMDP	AEC	EFG
Single Agent	★	★	✓	✓	✓	✓
Multiagent Cooperative	×	×	✓	★	✓	✓
Multiagent Competitive	×	×	✓	×	✓	✓
Multiagent Mixed	×	×	★	×	★	★
Multiagent Simultaneous	×	×	★	★	∼	∼
Multiagent Turn-based	×	×	∼	∼	★	★
Partial Observability	×	★	✓	✓	✓	✓
Full Observability	★	✓	✓	✓	✓	✓

Note that this is merely the base form of the theorem, and various modifications are possible, most notably in the form of **importance sampling** [Rub81], or adding a baseline to the reward $R(\tau)$. Some of these are discussed in the context of specific algorithms that use them in Section 4.

2.2.2. Bellman Equation

The Bellman Equation [Bel03] is a basis for all value-based algorithms. Unlike the Policy Gradient method, here we do not learn a policy directly. Instead, we try to approximate a state value function $V(s)$ or a state-action value function $Q(s,a)$. The former estimates the expected reward that the agent will collect in the future, given that it is present in a given state s. The latter estimates the same quantity, but given that the agent will take the specific action a in the state s. Then, we use these functions to generate a policy by choosing the best action in a given state. With a state value function V, this requires access to the environment transition function, which is not necessary with a state-action value, where the policy is simply given by $a = \arg\max_{a'} Q(s,a')$.

The value function Q^π (or analogously V^π) associated with a policy π represents the expected total reward if the agents is in a given state s, takes a certain action a ($a \sim \pi(s)$ for the state value function), and then proceeds by following the policy π for the rest of the episode.

$$V^\pi(s) = \mathop{\mathbb{E}}_{a_t \sim \pi}\left[\sum_t \gamma^t r_t | s_0 = s\right] \tag{10}$$

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{a_t \sim \pi}\left[\sum_t \gamma^t r_t | s_0 = s, a_0 = a\right] \tag{11}$$

The Q (or V) values of different state-action pairs (states) are obviously not independent – they are in fact related via the transition function, which determines what state comes after them. This is formalized by the Bellman Equation, which defines the consistency criterion of a Q (or V) function (Equations 12, 14), and the optimal function Q^* (or V^*) (Equations 13, 15):

$$V^\pi(s) = \mathop{\mathbb{E}}_{\substack{a \sim \pi \\ s' \sim T}}\left[R(s,a) + \gamma V^\pi(s')\right] \tag{12}$$

$$V^*(s) = \max_a \mathop{\mathbb{E}}_{s' \sim T}\left[R(s,a) + \gamma V^*(s')\right] \tag{13}$$

$$Q^\pi(s,a) = \mathop{\mathbb{E}}_{s' \sim T}\left[R(s,a) + \gamma \mathop{\mathbb{E}}_{a' \sim \pi} Q^\pi(s',a')\right] \tag{14}$$

$$Q^*(s,a) = \mathop{\mathbb{E}}_{s' \sim T}\left[R(s,a) + \gamma \max_{a'} Q^*(s',a')\right] \tag{15}$$

The intuition behind these equations is that the value of a state is equal to the instant reward obtained at that state, and the discounted expected value of the following state – which also includes the value of the state after that (due to the recursive nature of the equation), and so on until a terminal state. The value of a terminal state is typically considered to be 0, however a different convention may be used in certain cases, e.g. if the episode timed out. It also induces a dynamic programming solution of MDPs through the Value Iteration algorithm [SB18]. It is however inapplicable or impractical for many modern problems with complex state and action spaces, and instead, the Bellman Equation is used as the source of a differentiable loss function for value-based algorithms, as we describe in detail in Section 4.

It is worth noting that by using a Q function estimator $\hat{Q}^\pi$, we can obtain an alternative formulation of the Policy Gradient Theorem. Indeed, as shown by Sutton et al. [SB18], we get the following expression for the policy gradient:

$$\hat{g} = \sum_s d^\pi(s) \sum_a \nabla\pi(a|s) \hat{Q}^\pi(s,a) \tag{16}$$

where $d^\pi(s)$ is the marginal state distribution under the policy π. This formulation does not use individual transitions, but instead relies on statistics of the policy's performance, and can thus be used as an alternative algorithm to estimate the policy gradient.

2.3. Reward Hypothesis, Discounting, Advantage

It is worth taking a closer look at the assumption underlying all Reinforcement Learning research, sometimes called the **Reward Hy-**

pothesis. It is formulated by Richard Sutton as "That all of what we mean by goals and purposes can be well thought of as maximization of the expected value of the cumulative sum of a received scalar signal (reward)" [SB18]. This is reflected in the described formalisms and equations by the inclusion of a reward function R, with the goal of agents being maximization of the total reward obtained over their lifetime. Some argue that just the reward signal is sufficient to represent any goals that intelligent agents might have [SSPS21], while others point out that certain objectives cannot be represented with a single scalar reward [Ale20]. That being said, as we focus specifically on Reinforcement Learning in this work, we do not consider alternative formulations – but it is possible that they will become more relevant in the coming years as the field continues to develop.

An important element related to the reward function is the **discount factor** mentioned in the description of an MDP in Section 2.1.1. It can be considered either as a property of the environment, or the learning agent, and while the two views are mostly equivalent from the optimization point of view, they have potential implications relating to the Value Alignment problem [Soa18]. If we consider the discount factor to be a property of the MDP, this is the real reward we want the agent to optimize, whereas otherwise, we really want to optimize the total reward, and discounting the rewards helps improve the training in some way, e.g. as a form of regularization [AMC20]. It can also impact the range of methods that we can use – when considered as a part of the learning agent, any arbitrary method of reward discounting can be used, including non-exponential methods such as hyperbolic [FGB*19] or truncated [LH11] discounting.

One issue with using the raw rewards/utility for training is that it is an absolute metric, with no a priori point of reference. If the agent only perceives a single timestep where a certain action a_0 leads to a reward of -1, this action's probability will be decreased as the value is negative. However, it could still be the optimal action if the counterfactual rewards due to taking other actions are even lower. Asymptotically, this is all balanced out due to the fact that decreasing the probabilities of other actions will necessarily increase the probability of a_0. To decrease the variance of gradient estimation, some algorithms use the notion of **Advantage** instead. This often results in more stable and efficient training. Intuitively, advantage measures how much a certain action is better (or worse) than expected. Given both the Q and V function approximations, we define the advantage as:

$$A(s,a) = Q(s,a) - V(s) \tag{17}$$

In practice, algorithms that use advantage often compute $Q(s,a)$ from collected experience, i.e. , look at the trajectory and compute the total reward, while $V(s)$ is approximated with a separate neural network. Examples of this are included in Section 4.

3. Classification of RL Algorithms

In this section we describe the main categories of modern RL algorithms. While the division is not clear-cut and many algorithms at least draw on ideas from other types, we nevertheless consider this classification to be useful for building an intuition of the RL algorithm landscape. A diagram classifying the algorithms described in this work is in Figure 2. The details of these algorithms are provided in Sections 4 and 5.

3.1. Policy-based or Value-based

The first axis of division is whether the algorithm is **policy-based (PB)** or **value-based (VB)**. Although the state-of-the-art algorithm often use both components via Actor-Critic architectures, oftentimes they still have one part that is dominant in the overall picture. The difference between PB and VB algorithms is in what the model is actually trained to predict. In pure PB algorithms such as REINFORCE [Wil92, SMSM99], the neural network is trained to directly output the **action** that will maximize the expected future reward. On the other hand, pure VB algorithms like Deep Q Learning (DQN) [MKS*15] train the network to instead output the **value** of each action in a given state, that is the expected future reward. This works in environments with a discrete action space, because a policy can then be generated by taking the action with the maximum expected value.

3.2. Actor-Critic

Very commonly, RL algorithms use the so-called Actor-Critic architecture, which involves training two networks. One, the **Actor**, also called the **policy**, is responsible for predicting the action that the agent should take, as in PB algorithms. The other, the **Critic**, is responsible for predicting the **value** of an action in a given state, as in VB algorithms. The outputs of the two networks, while not always in agreement with each other, can be used to improve the training process in ways that depend on the exact algorithm – for example, by using the value prediction as a baseline for advantage estimation as in PPO [SWD*17], or by training the Actor to find the action with the highest value predicted by the Critic in order to use value-based methods in continuous action spaces as in DDPG [LHP*15].

3.3. On-policy or Off-policy

Another point of difference between RL algorithms is the data used for the optimization process, which does not necessarily have to be obtained with the same policy that is being learned. We normally refer to the **target policy** as the policy that is being optimized and will be used for evaluation, and the **behaviour policy** as the policy used by the agent to select actions and explore the environment. In **on-policy** algorithms like REINFORCE, the neural network can only be trained using data collected with the policy that is being optimized, meaning that the behavior policy matches the target policy. This implies that after performing a single gradient update, the data (in theory) has to be discarded. On the other hand, in **off-policy** algorithms like DQN, any data (trajectories) can be used, regardless of how it was generated (target and behaviour policies can be different). Some algorithms like PPO toe the line between being on-policy and off-policy, by allowing a relatively small number of gradient updates before the data has to be discarded by using tricks like importance sampling and clipping the loss function. Nevertheless, these algorithms are typically considered to be on-policy, as they cannot use data collected by an arbitrary behavior policy.

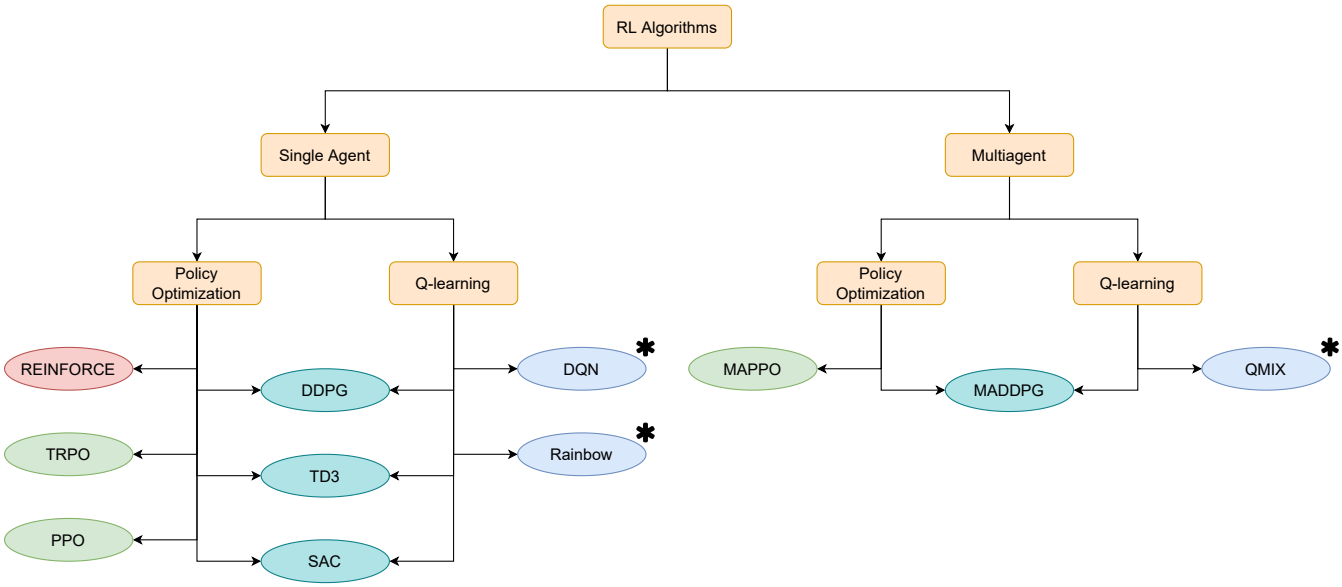

Figure 2: *A diagram showing a taxonomy of the Reinforcement Learning algorithms described in this work. We focus on two divisions: single agent or multiagent, and policy-based or value-based. The colors of nodes correspond to whether the algorithm is on-policy (red), off-policy (blue), or in between (green). Algorithms marked with an asterisk (∗) can only be used with discrete action spaces.*

Typically, on-policy algorithms use a **rollout buffer** which stores the environment transitions collected with the current policy, and is emptied after performing the gradient update. Off-policy algorithms instead use an **experience replay buffer**, which stores older transitions, replacing the oldest ones once it reaches maximum capacity.

3.4. Model-free or Model-based

This division relies on whether or not the learning agent has access to a model of the environment $T(s,a)$. In **Model-free approaches** like DQN, PPO or DDPG, the agent learns in a true trial-and-error fashion – it has no way of "knowing" the consequences of an action until it tries it, and observes the outcome. On the other hand, **Model-based approaches** additionally learn a model of the environment, allowing the algorithm to do something akin to traditional planning algorithms by considering potential future states and actions, without actually having to execute them in the environment. This is famously present in the AlphaZero [SHS*17] algorithm that achieved superhuman performance in the game of Go, where one of the components is the Model-based Monte Carlo Tree Search (MCTS) [Cou06]. While Model-based approaches can provide an advantage in planning terms, the effectiveness of the agent will be limited by the quality of the learned model, which can be negatively affected if the environment is very complex, which is often the case in character animation. This is not the case with Model-free approaches, which do not require an accurate characterisation of the environment to be effective, although they lack the ability to explicitly foresee future states and actions. In this work, we focus on model-free algorithms due to their relevance to character animation.

3.5. Single-agent or Multiagent

Finally, an algorithm can be designed to work with either one agent, or multiple agents sharing the same environment. While most of RL development focuses on single-agent algorithms, those can be extended to become multiagent algorithms through Independent Learning (see Section 5.1). In competitive multiagent scenarios, algorithms typically use the concept of self-play, training against (possibly old) copies of themselves so that they can be robust when matched with a wide range of opponents. In cooperative scenarios, a common trend is introducing some type of centralization of information so that the agents can coordinate more effectively.

3.6. Summary

Looking at modern RL algorithms, it is difficult to cleanly separate them into different categories. Many of the most successful approaches combine different concepts, resulting in an algorithm that is, technically speaking, actor-critic and off-policy. That being said, if we are content with the definitions being fuzzy, we can still gain useful insights about the differences between them.

Typically, value-based algorithms are also off-policy, and enjoy higher sample efficiency compared to policy-based ones. This is because any environment transition, once generated, can be used in perpetuity in multiple gradient updates. Conversely, policy-based algorithms like PPO make it possible to perform fewer gradient updates, because they involve optimizing the objective function directly through gradient ascent. This indicates that value-based methods can be better when it is difficult to obtain additional data, whereas policy-based methods can often be trained with smaller hardware needs, as they require fewer network updates.

4. Single-agent RL Algorithms

In this section we provide descriptions of the most important modern RL algorithms. Due to the large quantity of different methods that appeared in the recent years, this is not meant to be a comprehensive list of all algorithms that could be applied in character animation, but rather the ones with the most relevance, either to this application in specific, or for the field in general. We also provide a sufficient amount of detail for the reader to grasp the main ideas of the algorithms, but refer them to the source papers for the remaining information. Specifically, we do not aim to include sufficient information that would make it possible to reimplement the algorithms without referring to the main paper or existing implementation, as that tends to be a very complex process, with many details being important.

4.1. DQN

The first algorithm we discuss is **Deep Q Network (DQN)** [MKS*15], which gained prominence when it was used to master a suite of Atari games, achieving superhuman performance in some of them, drawing significant attention to the field. It is a prime example of a Value-based, Off-policy algorithm, and is remarkably simple in its basic form, allowing for a plethora of modifications which we discuss further in this section. DQN is a modern version of the older Q-Learning algorithm [WD92] which relies on the same principles, but only works on tabular domains (i.e. with a finite number of states and actions).

In DQN, the agent is defined by a state-action value function $Q(s,a)$, represented with a neural network, which is then trained to approximate the real optimal Q function of the environment. This is achieved by performing gradient descent on a Bellman loss function, defined as

$$\mathcal{L}_i(\theta_i) = \mathop{\mathbb{E}}_{s,a\sim\rho(\cdot)}\left[\left(y_i - Q(s,a;\theta_i)\right)^2\right] \tag{18}$$

$$y_i = \mathop{\mathbb{E}}_{\substack{s,a\sim\rho(\cdot)\\ s'\sim T(s,a)}}\left[R(s,a) + \gamma \max_{a'} Q(s',a';\theta_{i-1})\right] \tag{19}$$

where i is the current training iteration, θ_i are the weights of the neural network, and ρ is the probability distribution over state-action sequences according to the behavior policy. Intuitively, the network is trained in a way similar to supervised learning, with the target being the empirical Q value of a given state-action pair, obtained by executing the policy and estimating the future utility using the same current Q function estimate. Typically, an automatic differentiation software is used to find the gradient of the loss function with respect to the network weights θ, leading to the actual parameter update proportional to $\nabla_\theta \mathcal{L}(\theta)$

In order to ensure sufficient exploration, DQN uses ε-greedy sampling. This means that given a value of $\varepsilon \in [0,1]$, then while collecting data for optimization, the agent will choose a random action with a probability of ε, and the optimal action (according to the current Q function estimate) with a probability of $1-\varepsilon$. Commonly, ε is treated as a constant during a single training iteration, and progressively reduced to 0 as the training proceeds.

DQN also uses a replay buffer – the collected data is stored and reused throughout the training, which is possible because DQN is an Off-policy algorithm. So the general flow of the algorithm is as follows. First, collect a batch of data using the current behavior policy (defined by the weights θ_i and some value of ε), and add that data to the persistent replay buffer. Then, sample some data from the replay buffer, and perform gradient updates according to Equation 18. Repeat this process, updating the weights and decreasing ε until convergence.

A crucial limitation of the DQN algorithm lies in the max operator of Equation 19. With a discrete action space, finding the optimal action is easy – simply evaluate the function for each action, and then choose the best one. However, when dealing with continuous action spaces, this turns into a potentially non-trivial and nonlinear optimization problem, which in unfeasible to solve each time the agent needs to choose an action, which means that effectively, DQN is limited only to discrete action spaces. This can be avoided by changing the action space through discretization, or changing the algorithm (see Section 4.8).

4.2. Rainbow

Over the last few years, many modifications of the core DQN algorithm have been developed, aiming at various improvements to its performance. Six of them were combined in the **Rainbow** [HMvH*17] algorithm:

1. Double Q-Learning [vHGS15]
2. Prioritized Experience Replay [SQAS16]
3. Dueling Networks [WSH*16]
4. Multi-step Learning [Sut88]
5. Distributional RL [BDM17]
6. Noisy Nets [FAP*17]

The main ideas of them are as follows. **Double Q-Learning** trains two neural networks, decoupling the action selection from evaluation, in order to mitigate the problem of the learned Q networks overestimating the utilities. **Prioritized Experience Replay** changes the way in which old experience is sampled to optimize the Q network, so that more informative samples (i.e. ones with large updates) occur more frequently. **Dueling Networks** have two computation streams, one for the value, and one for advantage, with some of the weights shared between them. **Multi-step Learning** involves a different way of bootstrapping the future rewards, by looking a few steps ahead (as opposed to just one). **Distributional RL** has the algorithm learn to predict the distribution of rewards, as opposed to just the mean reward itself. Finally, **Noisy Nets** improve exploration by using partially stochastic linear layers. For further details on each of these modifications, we refer the reader to the relevant papers.

Overall, Rainbow agents generally train faster and reach a higher performance than the baseline DQN agents. This comes at the cost of implementation complexity, with only some of the standard frameworks supporting it (see Section 9), whereas DQN is very common, and relatively easy to implement in its basic form even for beginners.

4.3. REINFORCE

Similarly to how DQN is the simplest Value-based algorithm, **REINFORCE** [Wil92, SMSM99] is the original Policy-based method that is used with neural networks as function approximators. In its simplest form, it is a direct implementation of the Policy Gradient Theorem (see Section 2.2.1). It involves training a neural network to directly approximate the optimal stochastic policy $\pi\colon \mathcal{S} \to \Delta\mathcal{A}$, so that the expected total reward is maximized. This process is performed in an On-policy manner, with a fundamentally simple basic training loop:

1. Execute the policy and collect a batch of experience.
2. Perform a single gradient update of the policy and discard the data.
3. Repeat (1) and (2) until convergence.

REINFORCE can employ some improvements to a naive implementation of the Policy Gradient Theorem. Recall the general policy gradient estimate:

$$\hat{g} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \tag{20}$$

While the reward $R(\tau)$ is computed for the entire trajectory, it is reasonable that when considering the action at a step $t_0 > 0$, we disregard the rewards obtained before, i.e. for $t < t_0$, since the action at t_0 could not have affected them. Furthermore, subtracting a state-dependent baseline from the reward does not change its expectation, which means we can use the **advantage** instead, as defined in Section 2.3. This is useful as it decreases the variance of the gradient estimation, leading to a faster and more stable training procedure.

The policy trained by REINFORCE is stochastic, which means that it outputs a distribution over actions $\Delta\mathcal{A}$ rather than a single action. During training, the agent samples an action from the distribution in accordance with the policy gradient theorem. During deployment, it might be desirable to use the deterministic optimal action (i.e. $\max_{a \in \mathcal{A}} \pi_\theta(\cdot|s)$) for improved stability and predictability of the agent. Typically, a stochastic policy with continuous actions is modeled by a Normal (or Multivariate Normal for multidimensional action spaces) distribution. The neural network then outputs the mean action μ, and the variance σ^2 under the assumption that the individual components of the action vector are uncorrelated. Alternatively, a global, state-independent variance can be maintained in the model, and adjusted during the training. In the case of discrete actions, the policy uses a Categorical distribution, with the neural network outputs corresponding to their logits. Mixed action spaces are also possible, and can be modeled as joint distributions.

REINFORCE, as well as the algorithms based on it, can be trained as Actor-Critic algorithms. The Actor is the policy network π_θ which is responsible for the actual decision making, while the Critic V_θ is trained using regular supervised learning techniques, and is responsible for the value estimation in computing the advantage.

4.4. TRPO

Trust Region Policy Optimization (TRPO) [SLA*15] is based on REINFORCE combined with the notion of a Natural Policy Gradient [Kak01]. It aims to improve the amount of utility that the agent can obtain from a single batch of data. Recall that REINFORCE can only perform a single gradient update with a batch of data, usually with a constant or decaying learning rate. If the learning rate is too large, a small change in the policy weights can have a large impact on the behavior of the agent, making it difficult to tune while still maintaining good training efficiency.

In TRPO, there are several approximations that deviate from the theoretically-justified REINFORCE algorithm, but instead enable better practical performance. The key idea is the **trust region**, which corresponds to a constraint on the allowed KL divergence between policies in consecutive training steps. The general (theoretical) TRPO update in the training step $k+1$ is:

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}(\theta_k, \theta) \tag{21}$$

$$\text{s.t. } \bar{D}_{KL}(\theta||\theta_k) < \delta \tag{22}$$

where δ is a hyperparameter defining the size of the trust region, and $\mathcal{L}$ is the surrogate advantage:

$$\mathcal{L}(\theta_k, \theta) = \mathop{\mathbb{E}}_{s,a \sim \pi_{\theta_k}} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A(s,a) \right] \tag{23}$$

which measures how the new policy performs compared to the old one. The most important feature of this approach is that theoretically, the KL divergence constraint ensures monotonic improvements with a sufficiently small δ, while still being more sample-efficient than REINFORCE.

Due to the arg max operator in Equation 21, each step is a constrained optimization problem, which is infeasible to solve hundreds or thousands of times throughout the training. For this reason, the actual algorithm uses additional approximations, resulting in an efficient, but complex Policy Gradient method. Due to this complexity, as well as the fact that other methods can be applied on minibatches of data and are more efficient (see: PPO, Section 4.5), TRPO is rarely used in practice.

4.5. PPO

Proximal Policy Optimization (PPO) [SWD*17] is the successor to TRPO, which through additional simplifications and approximations achieves comparable performance, but with a significantly simpler implementation. It is the de facto standard Policy Gradient algorithm at the moment, and is supported by all major libraries.

Its core idea is to take several gradient update steps with an importance sampling term, without making the policy deviate too far from the original behavior policy. There are two main variants of PPO: PPO-Clip and PPO-Penalty. The former introduces a clipping term to the relative action probabilities in order to disincentivize large policy changes, as measured by KL divergence. The latter

adds a penalty term to the loss function for the same effect. In practice, the PPO-Clip variant is more commonly used. Their respective loss functions are as follows:

$$L^{CLIP}(\theta) = \mathbb{E}\left[\min(r_t(\theta)A_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t)\right] \quad (24)$$

$$L^{KLPEN}(\theta) = \mathbb{E}\left[r_t(\theta)A_t - \beta \mathrm{KL}[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right] \quad (25)$$

where ε is a hyperparameter, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ is the probability ratio of the action, and β is a coefficient which is adaptively adjusted during the training (if using the Penalty variant).

PPO typically uses an entropy bonus to improve exploration. This means that there is an additional term in the loss function proportional to the entropy of the policy π_θ, resulting in the policy maintaining some randomness, even at the cost of efficiency.

PPO is an Actor-Critic algorithm, with the Critic being responsible for value estimation that is then used to compute the advantages. The Critic network is typically trained by performing gradient descent on a Mean Square Error loss function between its outputs, and the empirical returns observed in the collected data.

While PPO is typically considered as an On-Policy algorithm, that is not entirely accurate. A single PPO update typically involves several gradient updates, often performed on minibatches of experience, after which the data is discarded as is the case in REINFORCE. This means that while the data can be reused, it can only be done in a very limited way, unlike typical Off-Policy algorithms.

It is worth noting that policy-gradient algorithms (REINFORCE, TRPO, PPO), tend to be sensitive to the implementation details which we omit from this survey. This phenomenon is analyzed in three large-scale studies, to which we refer interested readers: [HIB*17, EIS*20, ARS*20].

4.6. A3C, A2C

The **Asynchronous Advantage Actor Critic (A3C)** [MBM*16] algorithm, and its synchronous equivalent **Advantage Actor Critic (A2C)** [WML*17] are largely of historical value now. The key idea of A3C is using multiple parallel copies of the environment, from which the data can be collected asynchronously, without needing to synchronize them between episodes, or between individual steps. This is meant to improve training efficiency by eliminating the time when an individual worker has to wait for the main process to collect their experience and perform a gradient update.

When researchers continued working with A3C, they discovered that the asynchrony was not a necessary component, but rather an implementation detail, so they developed a simplified, synchronous version named A2C. This algorithm, in its essence, is very similar to REINFORCE with specific details such as using multiple parallel copies of the environment, and using a learned baseline for advantage estimation (which is not the original intent of REINFORCE, but is nevertheless an option in it).

4.7. GAE

While it is not a Reinforcement Learning algorithm in the same sense that DQN and PPO are, **Generalized Advantage Estimation (GAE)** [SML*18] is a method that can be applied to any algorithms which use the notion of advantage. It is heavily based on the concept of TD-lambda [Sut88], and can be seen as its extension using Advantages. In the simplest sense, given a trajectory with rewards r_t and a value estimation at each step V_t, we define the **Monte Carlo** advantage as:

$$A_t = \sum_i \gamma^i r_{t+i} - V_t \quad (26)$$

which is to say that we compute the expected total reward obtained by the agent, and subtract its estimated value. To use this expression directly, we need a full episode, which in certain environments might be infeasible or inefficient. Furthermore, as the sum of rewards depends on many decisions that the agent has yet to take in the future, the variance of this advantage estimation tends to be very large.

An alternative way is using **Temporal Difference (TD)** estimation by bootstrapping the expected returns, using the value function itself. Like before, given the rewards r_t and value estimations V_t, we define the TD advantage, or one-step advantage, as:

$$A_t^{(1)} = r_t + \gamma V_{t+1} - V_t \quad (27)$$

With an unbiased value estimator, the expected value of this expression is the same as Equation 26. At the same time, the variance can be significantly lower due to the lack of direct dependence on future rewards. With a biased value estimate, this becomes an example of the classic bias-variance trade-off, prevalent in Machine Learning.

Notice that intermediate n-step advantages can be defined by simply delaying the bootstrapping:

$$A_t^{(2)} = r_t + \gamma r_{t+1} + \gamma^2 V_{t+2} - V_t \quad (28)$$

$$A_t^{(n)} = \sum_{i=0}^{n-1} \left[\gamma^i r_{t+i}\right] + \gamma^n V_{t+n} - V_t \quad (29)$$

which introduces a wide range of possible advantage estimation methods. What GAE proposes is using all n-step advantage estimates, weighted exponentially with a factor of $\lambda \in [0,1]$:

$$A_t^{GAE} = (1-\lambda)(A_t^1 + \lambda A_t^2 + \lambda^2 A_t^3 + \dots) \quad (30)$$

This turns out to have a simple analytic expression that can be computed with a single pass algorithm. Empirically, GAE often noticeably improves the performance of RL algorithms, and is the de facto standard for advantage estimation in Actor-Critic algorithms.

4.8. DDPG

An algorithm on the boundary between Value-based and Policy-based methods is the **Deep Deterministic Policy Gradient (DDPG)** [LHP*15]. It is based on the notion of a Deterministic Policy Gradient [SLH*14], which is the gradient of a state-action

value function with respect to the action. It can also be seen as an adaptation of the DQN algorithm to continuous action spaces.

In DDPG, we train two separate networks – a state-action value network $Q_\phi\colon \mathcal{S}\times\mathcal{A}\to\mathbb{R}$, and a (deterministic) policy network $\pi_\theta\colon \mathcal{S}\to\mathcal{A}$. The value network is trained in a way similar to DQN, with some tricks such as using a replay buffer and a target network to stabilize the training. The key difference lies in the max operator of Equation 19, which is not trivial with a continuous action space. This is where we use the second, policy network, trained to predict the optimal action according to the reward function. The Q network is optimized to minimize the following loss functions:

$$L(\phi) = \mathbb{E}\left[\left(Q_\phi(s,a) - y(r,s',d)\right)^2\right] \tag{31}$$

$$y(r,s',d) = \left(r + \gamma(1-d)\max_{a'} Q_\phi(s',a')\right) \tag{32}$$

where (s,a,r,s',d) are the transitions in the replay buffer, with s,s' being the current and next state, a the action that was taken, r the reward, and d is equal to 1 if the state was terminal, and 0 otherwise. Then, the policy is optimized using gradient ascent to maximize the following objective:

$$L(\theta) = \mathbb{E}\left[Q_\phi(s,\mu_\theta(s))\right] \tag{33}$$

This can then be differentiated using the chain rule, giving the policy gradient of:

$$\nabla_\theta L(\theta) = (\nabla_a Q_\phi(s,a))\cdot(\nabla_\theta \mu_\theta(s)) \tag{34}$$

Overall, DDPG can be seen as the simplest way of adapting DQN to continuous action spaces, without having to discretize the action space. Because it is off-policy, it can be more sample-efficient than competing on-policy algorithms, making it suitable for environments in which it is difficult to collect large amounts of data. However, its asymptotic performance is often worse than that of competing on-policy algorithms like PPO, which leads to its limited practical use in character animation.

4.9. TD3

Twin Delayed DDPG (TD3) [FvHM18] is to DDPG what Rainbow is to DQN – it introduces a series of tricks that significantly improve the algorithm's performance. The main changes are as follows:

1. Clipped Double Q-Learning
2. Delayed Policy Updates
3. Target Policy Smoothing

Clipped Double Q-Learning works similarly to Double Q-Learning described in Rainbow (Section 4.2, using the smaller value of the two networks' outputs to prevent value overestimation. **Delayed Policy Updates** involves performing policy updates less frequently than Q function updates. Finally, with **Target Policy Smoothing**, noise is added to the target action, so that it is more difficult for the policy network to exploit errors in the Q function.

4.10. SAC

Soft Actor-Critic (SAC) [HZAL18] is in many ways similar to TD3, in that it is a modification of DDPG with certain changes introduced in order to improve its performance. Primarily, it uses entropy regularization by adding a term proportional to the policy's entropy to its optimization objective. This encourages the policy to remain stochastic, increasing exploration. Similarly to TD3, it uses Clipped Double Q-Learning, minimizing the Bellman loss of DDPG. However, there is no explicit policy smoothing, as SAC trains a stochastic policy instead of a deterministic one. As a result, the additional regularization is unnecessary, since actions are sampled from a nontrivial distribution.

Specifically, SAC learns three functions in parallel: the policy π_θ, and two Q functions Q_{ϕ_1}, Q_{ϕ_2}, with the usual double Q-learning approach. Since they are trained on an entropy-regularized objective, the Q function optimization objective takes the following form:

$$L(\phi) = \mathbb{E}\left[\left(Q_{\phi_i}(s,a) - y(r,s',d)\right)^2\right] \tag{35}$$

$$y(r,s',d) = r + \gamma(1-d)\left(\min_{j=1,2} Q_{\phi_j}(s',\tilde{a}') - \alpha\log\pi_\theta(\tilde{a}'|s')\right) \tag{36}$$

where $\tilde{a}'\sim\pi_\theta(s')$, and $\alpha>0$ is the entropy regularization coefficient. Notice the similarity to Equations 31 and 32 of DDPG, with the key difference being that the objective now has a term proportional to the entropy of the action distribution $\alpha\log\pi_\theta(\tilde{a}'|s')$, and the action used for computing the Q value of the following step is taken directly from the behavior policy.

When it comes to policy learning, as SAC learns a stochastic policy, it must output a distribution over the action space. The optimization takes the following form:

$$L(\theta) = \mathbb{E}\left[\min_{j=1,2} Q_{\phi_j}(s,\tilde{a}') - \alpha\log\pi_\theta(\tilde{a}'|s)\right] \tag{37}$$

where $\tilde{a}'\sim\pi_\theta(s')$. Notice again the similarity to Equation 33, which confirms that SAC is, in its essence, an updated and improved version of DDPG.

It is important to keep in mind that while this is a general outline of the algorithm, there are many details that can significantly affect its performance. For more information on this, we refer the reader to the original paper, as well as the existing open-source implementations (Section 9.3).

4.11. Learning from Data

As a general rule, Reinforcement Learning does not need expert data to train agents, instead using an environment that the agent

can interact with. In some cases, however, it may be beneficial to use expert data to augment the learning process, or even eliminate the use of a simulation whatsoever. This is often referred to as **Imitation Learning**, because the agent learns to imitate the actions of an expert whose experience is shown to it.

Behavior Cloning (BC) [BS99,RGB11,DBH16] is the simplest way to perform Imitation Learning. Its core idea is to treat Imitation Learning as a supervised learning problem, which given a dataset consisting of observations and actions, learns to map the former to the latter by training a classifier or a regressor.

By including a model training phase in which the agent can interact with the environment, we can remove the requirement that the dataset contains the actions [TWS18]. This significantly simplifies the data required to perform imitation learning, and enables learning by simply observing someone, much like humans do in the real world. However, the quality of the resulting behaviors is typically lower due to the fact that the dynamics model is only trained with on-policy data, which means that out-of-distribution errors are likely to occur. For this reason, if the data about actions is available, it is better to use it instead of relying only on observations.

The **Generative Adversarial Imitation Learning (GAIL)** [HE16] algorithm represents the main alternative to Behavior Cloning. It relies on the concept of **Inverse Reinforcement Learning (IRL)** [ZMBD08], which means learning the reward function from demonstrations (as opposed to regular RL, where the agent learns a policy, or generates demonstrations, given the reward function). This, combined with the notion of adversarial learning known from **Generative Adversarial Networks (GAN)** [GPAM*20], and a PG-based update rule (originally TRPO) produces an efficient algorithm for Imitation Learning.

A common practice is using Imitation Learning methods in conjunction with standard, reward-based RL algorithms [GGL*20]. This can be done by including a term derived from Imitation Learning either in the reward function, indirectly encouraging the agent to act similarly to the data, or by including it directly in the optimization objective.

4.12. Summary

We described the most noteworthy RL algorithms used in single-agent environments. From a practical point of view, we recommend either using the on-policy **PPO** with GAE for advantage estimation, or the off-policy **SAC**, which are the most popular algorithms of their respective categories. If the training data is difficult to obtain, SAC is typically better as it can reuse the data enabling higher sample efficiency. On the other hand, PPO often offers faster training in terms of the wall time by using parallelism in data collection and larger performance improvements per gradient update. If working with discrete actions, Rainbow or another version of DQN is also a viable choice. Finally, if one wants to incorporate real data in the training process, both BC and GAIL are strong options and can be integrated with other algorithms.

5. Multiagent RL Algorithms

Here, we describe the algorithms that are adapted specifically for multiagent environments. Those are typically based on existing single-agent algorithms, with modifications that improve the training process by abusing the specific multiagent structure of the problem.

5.1. Independent Learning

Any single-agent algorithm can be used in a multiagent scenario by using **Independent Learning**, with the resulting algorithms typically called **IPPO**, **IDDPG** etc. This entails treating the other agents as parts of the environment, possibly including information about them in the observation, and then simply training as if it were a single-agent task. A simple way to accelerate this training process when all agents are identical is treating them as **homogeneous**, also called **Parameter Sharing** [TGH*20]. With this approach, every agent receives their own observation and takes their own action, but they share the underlying neural network, and their experience can be combined for the training. Otherwise, if each agent has its own separately trained neural network, it is referred to as **heterogeneous**. It is possible to introduce some degree of heterogeneity by including an agent indicator in the agent's observations, as shown by Gupta et al. [GEK17].

5.2. MADDPG

MultiAgent DDPG (MADDPG) [LWT*20] is an extension of the DDPG algorithm to explicitly use the structure of multiagent environments in the training procedure. It relies on the idea of **Centralized Training, Decentralized Execution (CDTE)**, which means that the algorithm can use global or hidden information, as long as the resulting agent only needs access to its own observations.

In multiagent environments, there are two main pieces of information that is not available during execution – other agents' observations (or the global state), and the actions they take. However, when training in a simulation that we have total control over, these quantities are readily available, and so can be used in an Actor-Critic paradigm to optimize the Critic. Then, in the execution phase, only the agent's local observation is necessary for the Actor network to choose the action.

5.3. MAPPO

MultiAgent PPO (MAPPO) [YVV*21] is the result of extending the PPO algorithm analogously to the difference between DDPG and MADDPG. Because PPO is an Actor-Critic algorithm, the Critic similarly use centralized information such as other agents' observations and actions, while only the Actor is actually involved in the decision making during evaluation.

Since the concept and the name of MAPPO is generic, there are other works that introduce a similar extension [GTZL20,LWL*20, HHL21]. The details are different between those papers, but in the most robustly evaluated version of it uses the following five tricks:

1. Value normalization through a running mean, for robustness with respect to the reward scale
2. Value function input includes both global and agent-specific features, pruned to reduce the input dimensionality
3. Data is not split into minibatches, and the algorithm uses relatively few training epochs

4. The clipping factor is tuned as a trade-off between training stability and fast convergence
5. Using death masking (inputs for dead or deactivated agents) through zero states with agent ID

The resulting algorithm delivers results comparable with more sophisticated off-policy algorithms, while being viable to train using a single machine with one GPU.

5.4. QMIX

QMIX [RSdW*18] and its derivatives are a family of algorithms that adapt Q-learning in cooperative scenarios, so that it can use centralized training, while maintaining the option to perform decentralized execution. The core idea is that the joint state-action value is a monotonic function of the state-action values of each individual agent.

Consider the two extremes in terms of centralizing Q value estimation. On one hand, we have fully independent Q learning, where each agent optimizes their own reward, which can be a viable option as described in Section 5.1. On the other hand, we can also consider fully centralized Q learning, with a single network processing all agents' observations, and outputting their joint action. A simple middle ground can be found in Value Decomposition Networks (VDN) [SLG*17], where a joint Q function is expressed as a simple sum of the agent's individual Q functions:

$$Q_{tot}(s,a) = \sum_i Q_i(s^i, a^i) \tag{38}$$

QMIX introduces additional flexibility to this approach. It replaces the summation operator with an arbitrary function of the individual values, with the only restriction being that it is monotonic with respect to all its inputs:

$$\frac{\partial Q_{tot}}{\partial Q_i} \geq 0, \forall i \in \mathcal{I} \tag{39}$$

This is obtained by using a **mixing network** to represent Q_{tot}. The weights of the mixing network are the outputs of a set of hypernetwork [HDL16] conditioned on the environment state. This whole setup can be trained with significant information sharing between the cooperating agents, while in the execution phase, each agent only requires its own Q function Q_i.

Due to the popularity and effectiveness of QMIX, researchers have developed various modifications aimed at improving its performance even further [ZLS*20, WRL*21, YHL*20, RFPW20, SKK*19]. However, recent work suggests that using regular QMIX with appropriate implementation details is enough to achieve results comparable or even superior to the more complicated algorithms [HWH*21].

5.5. Summary

When working with multiagent problems (e.g. crowd simulation), we typically recommend using one of the single-agent algorithms and applying it with an **Independent Learning** approach as a starting point, with either **IPPO** or **ISAC** following the notation from Section 5.1, as well as **Parameter Sharing**. This is significantly simpler in implementation than using algorithms that introduce centralized communication, and can often yield competitive results. While adding some additional communication or centralization may be beneficial, MADDPG tends to be difficult to train in new environments.

6. Skeletal Animation

Individual characters can be animated using kinematic or physics-based methods. For the former case, the action space directly consists of kinematic poses or existing motion clips, and are defined based on motion capture data. In contrast, physics-based methods have action spaces that directly or indirectly produce joint torques that drive the motion. In this section, we first provide an abridged overview of RL as applied to kinematic methods. We then shift our focus to physics-based methods. This begins with a general summary of the many nuances involved when using RL to control physics-based character movement, given that the default motions produced by RL algorithms for humanoid characters in the RL literature are usually of low quality as compared to what is needed for computer animation applications. We then categorize and review many of the recent methods and results for RL-based physics-driven character animation.

6.1. RL for Kinematic Motion Synthesis

RL has a long-standing history of being used to learn kinematic controllers from motion capture data. Here we provide a brief overview of work in this direction. Motion generation can be framed as an RL problem where actions correspond to the choice of motion clips, as first applied to automatically-constructed graphs [AF02, KGP02, LCR*02] and then in ways that were better tailored to locomotion tasks, e.g., [LL06, TLP07]. Lee et al. [LWB*10] later introduced the concept of continuous motion fields in support of a data-driven state-dynamics model. Optimal actions on this model are then learned via a table-based representations for the policy and value function. Modern motion matching methods can be seen as a short-horizon version of motion-fields. Ling et al. [LYZ*20] learn a latent action space using autoregressive variational autoencoders to define character controllers and thereby enabling optimal goal-based animations.

6.2. The Many Challenges Beyond the Choice of Algorithm

A considerable amount of thought is typically required to define a character movement task, articularly in a physics-based setting. This begins with the design of the character, which involves making decisions related to joint torque limits, contact friction, mass distribution, joint limits, joint damping, simulation and control time steps, and more. The choice of action space can also have implications for the learned results. Available options include joint torques, joint PD-target angles, joint PD-target angle offsets from an available reference motion, muscle-based activations, or more abstract actions for hierarchical control approaches. It is also sometimes

possible to learn simplified actions spaces that avoid redundancies or that sample from a reduced-dimensionality action manifold, which can possibly be learned as well. The definition of the state of a character that is provided to the policy can also have a significant impact. The pose can be represented as Cartesian joint locations, or in a more traditional form consisting of a root position and orientation, followed by a set of internal joint angles. Contact information can also be an important part of the state.

Next, the task rewards need to be designed, which may need to balance generic and possibly temporally-sparse rewards related to the goals, rewards that encourage energy-efficient behavior, and shaping rewards that help guide the solution in what can otherwise be an exceedingly-large search space. Rewards also tend to work better when mapped to a fixed range, as commonly done using a negative exponential. Episode termination criteria are also important, as they effectively constrain the search space and, by virtue of providing no further rewards, also provide an implicit negative task reward. Reward terms can be combined, using a weighted addition, e.g., [PBYVDP17] or in a multiplicative fashion, e.g., [PRL*19], and these choices can strongly impact the final learned policies.

The optimization criteria to define a natural human or animal motion are difficult to determine, and thus a natural alternative is to instead seek to imitate motion capture data, either as individual motion sequences, or as distributions using adversarial approaches. The choice of initial states for a task is important, as it can affect the task difficulty [RTvdP20], and can also simplify the learning, as in the case of a motion imitation task where the initial states can be drawn from the given reference trajectory [PALvdP18]. Curriculum-driven learning can enable an easy-to-difficult learning order for a task [XLKvdP20]. Policies can be "warm-started" from existing solutions. Prior knowledge should be used where possible to set the relevant variances and exploration rewards. External forces can also be allowed early on in the optimization [YTL18], and then slowly withdrawn. Hierarchical learning can also be leveraged, by first learning low-level control that operates at a fine time scale, followed by higher-level control that allows for long-horizon tasks [PBYVDP17].

The algorithms themselves are challenging to work with, with a typical improve-and-test debugging iteration requiring between hours and days, depending on the task difficulty and the availability of compute. In many cases, wall-clock time is a more important consideration than sample-complexity, and algorithms whose common implementations support a high-degree of parallelization, e.g., PPO, are then sometimes preferred over that are more difficult to parallelize, e.g., SAC. Tuning the algorithm hyper-parameters plays an important role in the learning efficiency and success, and may require grid search or other hyper-parameter optimizations. The results of model-based trajectory optimization can be used to guide policies towards suitable solutions for difficult tasks. Debugging RL tasks is also an important skill, and points to initially working with simplified or more-constrained systems, visualizing reward terms, understanding the limitations of physics engines, and much more. More specific algorithmic features to consider include the use of a Huber loss instead of the conventional quadratic loss for Q-learning, considering various forms of conservative Q-learning, choice of temporal-difference horizon, and more.

Many simulated robotic control environments are standard benchmarks for RL algorithms. MuJoCo [TET12b] and PyBullet [CB16], two of the most commonly used physics simulation engines in RL, provide several robot models with a Gym [BCP*16] interface. These robots range from abstract ones like Hopper or Reacher, through animal-like Ant and HalfCheetah, to more human-like ones like Humanoid and Walker2d. While not realistic, they share many of the principles of skeletal character animation.

We next review work that uses reinforcement learning to develop a variety of full-body motion skills for physics-based characters. These leverage many of the insights described above.

6.3. RL for Individual Character Skills

For the remainder of this section, we further categorize methods into: (a) those which use motion capture data, typically as a key part of the imitation objective, and (b) methods that use a more general "pure" learning objective. In both cases, there exists a variety of prior art that is entirely model-based or uses other optimization methods. However, for our purposes here, we restrict ourselves to methods that use reinforcement learning for motion imitation.

6.3.1. Motion imitation RL methods

One of the first RL methods to be able to successfully imitate motion capture data, including highly dynamic motions such as flips, uses data from a stochastic planning method, first developed as an open-loop trajectory optimization method [LYvdP*10]. Building on this type of method, the work of [LPY16] proposed to use data from multiple runs of the stochastic trajectory optimizer to then learn a state-conditioned feedback policy. The desired motion sequence is divided into a sequence of 0.1 s duration control fragments, and for each such fragment computes a multivariate linear regression of the actions with respect to the state. This yields a simple linear policy for actions as a function of the state, for the duration of the control fragment. This model is then able to robustly imitate walking, running, spin-kicks, and flips, as well as transitions. Further work has then shown how learned control fragments can be treated as abstract actions, which can be resequenced using deep Q-learning [LH17], and can further be adapted to learn basketball playing skills [LH18].

The use of policy gradient RL methods to imitate human motion capture clips was first explored by Peng et al. [PBYVDP17] for a variety of walking gaits. This also introduced a hierarchical reinforcement learning approach, with a low-level policy first being trained to reach target stepping locations while also striving to imitate the reference motion. A high-level policy then operates once for every walking step, generating step targets in support of tasks, including control of a ball with the feet, navigating paths, and avoiding dynamic obstacles. Peng et al. [PALvdP18] further develop the imitation learning approach to train controller for a diverse set of motions, including highly-dynamic spin kicks and flips for humanoids, sequencing such motions, and using the same imitation approach for quadruped controllers. Imitation-based learning of a wider variety of quadruped gaits, including sharp turns, is demonstrated in [PCZ*20], along with successful transfer to quadruped robots. Peng et al. [PMA*21] use ideas from adversarial imitation learning by combining a reward function to control the

high-level behaviors, with low-level controls specified with an unstructured dataset of motion clips. This method can be used on both humanoid and non-humanoid models. It produces high-quality animations that match tracking-based methods, but the training process can still be prone to mode collapse, as is common in GAN-like algorithms. Some of these examples in Imitation Learning are shown in Figure 3. The choice of action space is also shown to have an impact the speed and quality of imitation-based learning [PvdP17].

Computer vision based pose tracking can also be used as a source of motions to imitate, allowing robust control policies to be learned from video clips [PKM*18]. Isogawa et al. [IYOK20] construct an end-to-end pipeline that converts Non-Line-Of-Sight measurements to 3D human pose estimation by employing a diverse set of techniques, including an RL-based humanoid control policy. Yuan et al. [YWS*21] introduce the SimPoE framework, which trains an RL agent to control a physics-based character to estimate plausible human motion, while conditioning it on a monocular video.

The majority of the works described above develop control policies that only reproduce single clips, or a specific set of motion clips. The motion to imitate plays a role via the reward, but is not provided to the policy as an input. The policies are conditioned on a time or motion phase. An important next step has been to reproduce a richer variety of motions by conditioning the policy on a short time window of the future motion to imitate. This can also be seen as a generalized form of learned inverse dynamics, with a longer anticipatory window as needed to make motion corrections for more difficult motions. Chentanez et al. [CMM*18] first develop this type of conditioning and apply it to large motion datasets. Significant further developments follow from improvements that target scalability, motion transitions, motion quality, generalization, and learning efficiency [PRL*19, BCHF19, WGSF20, WGH20]. These methods are further extended to work with muscle-based actuations [LPLL19], a large diversity of body shapes [WL19], and producing large motion variations even from a single motion clip [LLLL21]. Other work shows how to allow for more flexible forms of imitation [MYT*21], and that leverage residual external forces to enable learning more challenging motions [YK20]. Imitation-based controllers can also be used to learn a latent human-like action space via distillation ("neural probabilistic motor primitives"), which can then be used as an abstract action space for new tasks [MTA*20]. Similarly, Luo et al. [LSCC20] learn a natural action distributions from reference motions for quadrupeds, while a GAN-based controller reproduces suitable actions based on user-input. This is followed by high-level DRL fine-tuning.

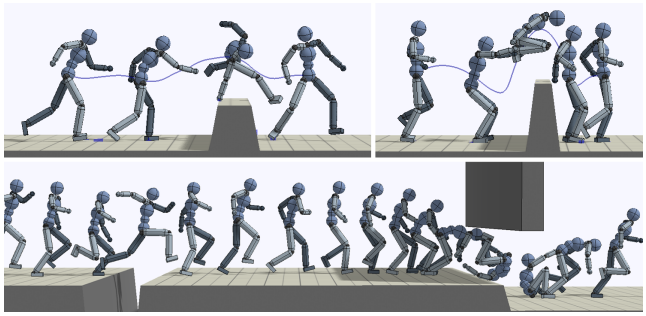

Figure 3: *Imitation-based Learning. Proposed methods as in [PALvdP18] allow to successfully synthesize animations from motion capture data. In other works, as in [PMA*21], they combine such techniques with the possibility of adding low-level behaviours to control the production of high-complexity animations.*

6.3.2. Pure objective RL methods

Reinforcement learning has also been successfully used for full-body character animation without an imitation objective. Here, the objective can be framed in terms of rewards that include energy, progress towards a goal, stylistic hints, and regularization terms.

Model-predictive control (MPC) methods, which iteratively replan and then execute the first action, have been successfully employed for humanoid animation and are a form of model-based RL. The work of Tassa et al. [TET12a] demonstrated the online use of iLQG (Iterative Linear Quadratic Gaussian) trajectory optimization for online control of humanoid characters for a variety of tasks, including getting up, using a 0.5 s planning time horizon. Sampling-based methods can also be used to achieve trajectory optimization over a finite planning horizon, and have been explored in detail by Hämäläinen et al. [HET*14, HRL15]. Online trajectory optimization and policy learning can also be used in a mutually supportive fashion [RH17], with the policy serving to accelerate the trajectory optimization, and the trajectory optimization helping to bootstrap the policy learning. In addition, trajectory optimization can benefit from more complex search spaces, for instance by including contact points [MTP12] to improve simultaneously both, trajectory and policy learning.

Actor-critic methods for RL can more easily tackle motion tasks, such as locomotion, by being provided with task-specific action abstractions. For example, the action space can consist of a discrete set of existing controllers, with a high-level actor-critic controller being trained to make a discrete selection among the set of available controllers at each step of the gait. This setup is used with a *k*NN-based value function approximator to achieve high-level objectives by Coros et al. [CBVdP09]. An abstracted, tailored action space is used by Peng et al. [PBvdP15] to include a continuous action space as defined by a conveniently-parameterized finite-state machine controller. A *k*NN-based actor-critic pair is then used to train dog-like and bipedal models to traverse variable terrain. Later, Peng et al. [PBvdP16] develop a mixture-of-experts based Actor-Critic algorithm named MACE for improved performance on a similar dynamic locomotion task, this time using deep neural networks for the actors and critics, and thereby eliminating some of the feature engineering required by the previous approach.

Can policy-gradient RL algorithms be used with pure learning objectives to generate natural human movement, as opposed to the unrealistic frenetic motions commonly seen resulting from popular RL benchmarks? Yu et al. [YTL18] encourage symmetric and low-energy motions by appropriately modifying the loss function of the algorithm, by adding the so called mirror-symmetry Loss to the usual surrogate loss of PPO. This allows for high-quality motions without using any imitation of motion examples. This is particularly important for non-humanoid characters for which there is no

motion data available. Example non-humanoid models that can be trained this way are in Figure 4. Abdolhosseini et al. [ALX*19] further investigate multiple methods of incorporating symmetry constraints for skeletal animation tasks, inspired by the observation that human and animal gaits found in nature are typically symmetrical. They use the PPO algorithm with four options of enforcing symmetry on the learned policy. They show that this can in fact be harmful to the training process, but in the end can produce higher quality motions. Xie et al. [XLKvdP20] explore a curriculum-based learning solution to train characters to walk and run over a wide range of stepping stones, with varying step heights, lengths, yaw angles, and step pitches, in the absence of an imitation objective. PPO is used in conjunction with a parameterized generator of individual steps, and the learning curricula advance the step difficulty in several different ways. PPO is used to train the physical legged model.

To further improve on the realism, more biomechanically accurate models can be considered. The NeurIPS conference hosted three challenges using the osim-rl platform (see Section 9.2): "Learning to Run" (2017), "AI for Prosthetics" (2018) and "Learn to Move - Walk Around" (2019) [KMO*18a], all of which dealt with different aspects of controlling a human body model. The leading solutions used learned models of the environment, and off-policy algorithms such as DDPG. Jiang et al. [JVWDGL19] continue this research direction by using another human body model based on the OpenSim [SSRD11] platform. With muscles outnumbering joints, the larger action space of biomechanical models can be significantly more expensive to train. This paper addresses this by allowing the optimization to nevertheless operate in joint actuation space, as afforded by two neural networks that model the state-dependent torque limits and the metabolic energy. The overall approach is agnostic to the choice of RL algorithm.

Non-locomotion tasks are also important for full-body character animation. Kumar et al. [KHL17] use an algorithm based on MACE for the task of teaching a virtual humanoid model to safely fall by minimizing the maximal impulse experienced by its body. They train a mixture of actor-critic networks associated with all possible contacting body parts, and further use a form of hierarchical reinforcement learning, with an abstract policy deciding the high-level behavior, and a joint policy responsible for actually executing the action. Clegg et al. [CYT*18] consider the problem of simulating the movement of a human dressing themselves using a combination of physics simulation and RL. They use a virtual human-like model and is tasked with putting on one of three different pieces of clothing. The process of getting dressed is divided into several subtasks, each of which is treated as a Reinforcement Learning problem with an appropriate reward function. Subtasks are trained using TRPO, and then sequenced together to produce a full motion. Yin et al. [YYVDPY21] explore the problem of learning diverse jumping motions, including high jumps. For a given takeoff state, a curriculum is used to learn a policy for increasing bar heights. The space of takeoff states is then explored using Bayesian diversity search, to synthesize a diverse set of jumping styles, including jump well-known techniques such as the Fosbury flop.

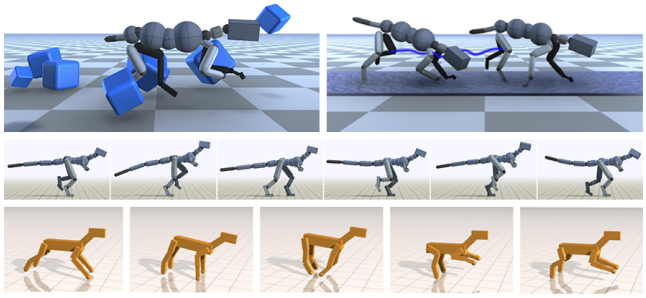

Figure 4: *There is a wide variety of methods that also address the synthesis of animations for quadrupedal or arbitrary morphology [LSCC20] [PALvdP18] [YTL18]. While the limited amount of motion capture data introduces an additional challenge, such methods try to overcome this constraint by covering a wide range of techniques, from imitation-based approaches to pure objective RL.*

7. Crowd Animation

In this section we discuss the specific work that used RL algorithms in Crowd Animation scenarios, as well as the challenges that make this task distinct from single-agent cases. Unlike Skeletal Animation, Crowd Animation typically uses multiagent algorithms so that each individual agent has access to its own information, but not necessarily to the global state. While the task can often be seen as fully cooperative (e.g. making a realistic simulation), this enables more realistic behaviors, and not using a centralized controller enables easier scaling to different numbers of agents. We focus on applications pertaining to the challenge of multiple agents navigating in a shared environment, and omit the discussion of more advanced topics around coordination and division of tasks, considering them to be out of scope of this work.

7.1. Challenges of Crowds

The key factor distinguishing cooperative multiagent learning from single agent RL is the phenomenon of nonstationarity. Typically, RL algorithms assume that the environment is stationary, which means that the environment dynamics (represented by the transition function) remain the same throughout the training. That is not true in multiagent training as observed by a single agent – as other agents learn, their policies change, which affects the perceived environment dynamics.

In the case of Crowd Simulation specifically, there is also the question on how exactly to represent the physics of the problem. While some works use holonomic cartesian controls in which each agent can move in any direction, this is not entirely realistics, and instead, polar controls may be used, where an agent decides its linear motion and turning left or right. Furthermore, the agents may either control their velocities directly, or apply accelerations to their motion. While these approaches can be seen as nearly equivalent, it is still necessary to choose one, which may impact the final performance in nontrivial ways.

In certain cases, competitive and general-sum scenarios may be relevant, which carries additional complications. Most notably,

evaluation of trained systems is challenging in the absence of an expert model or an external performance measure. This is because the typical training paradigm relies on self-play, and a winrate against a copy of itself cannot be reliably translated to objective performance. Furthermore, the details of the self-play procedure can also impact the training. Finally, as training progresses, agents might learn to specialize to play against specific strategies, forgetting about their older versions, and underperforming when matched up against them.

Finally, it is worth mentioning that many of the challenges in Skeletal Animation, still apply here. Depending on the physical model of the agent behaviors and interactions, the exact choice of the actuator may be important for the learned policies. Similarly, designing the reward function is crucial for good performance – a reward that is too sparse may be prohibitively difficult, whereas one that is adapted to be more dense, may lead to unexpected behaviors. Finally, it is often desirable to have agents exhibit human-like behavior, but this task in itself is not well-defined, and it may be helpful to use real-world data in order to generate a specific reward function.

7.2. Applications

Long et al. [LFL*18] apply an RL-based approach to the task of collision avoidance. While this is not exactly the same thing as character crowd animation, collision avoidance is nevertheless a significant component of crowd simulation systems. They train a policy which receives as input a depth map, the goal's relative position and the current velocity, on the task of reaching the goal and avoiding collisions with other agents in the shared environment. Because they also deploy them on real robots, there is additional emphasis on avoiding collisions, with contact between two agents leading to removing both of them from operation with a large negative reward. They train the policy using the PPO algorithm, and show that this method results in higher success rates and more efficient policies compared to ORCA. With a similar approach, Lee et al. [LWL18] apply the DDPG algorithm to the task of basic crowd simulation – a number of agents in a shared environment move through it, and attempt to reach their respective destination. They use polar dynamics, with the RL agent setting a linear velocity and a rotation at each timestep. The agents receive a positive reward for getting closer to their goals, a negative reward when they collide, and there is also a regularizing reward term that encourages smooth movement. They show that this setup is sufficient to obtain agents that reach their goals and avoid collisions, although the results are still imperfect. There is also no regard for human-like behavior.

To explicitly combine ORCA with RL methods on the same standard crowd simulation task, Xu et al. [XHLL20] introduce the ORCA-DRL algorithm. They use PPO to predict a preferred velocity at any given step, which is then used as an input to ORCA. This is then responsible for actually avoiding collisions. They show that this approach leads to successful collision avoidance, which is not surprising given its reliance on a classical collision avoidance algorithm. It also does not consider whether the behavior is human-like. Sun et al. [SZQ19] use a different approach – they train four "leader" agents responsible for guiding parts of the crowd, while the remaining agents follow their respective leaders. They use PPO with a recurrent LSTM [HS97] policy and combine it with a classical collision avoidance algorithm RVO. They train the agents to act in an unknown environment with dynamic obstacles. The resulting behaviors manage to achieve their goals, but do not take into consideration whether the behavior is human-like. This method also still relies on classical collision avoidance algorithms.

Haworth et al. [HBM*20] introduces a method on the borderline between single character and crowd animation. By employing a method based on the ideas of Hierarchical Reinforcement Learning, they train two policies interacting with one another. The high-level policy is responsible for navigation and reaching global goals. It sets objectives for the low-level policy which directly controls the joints of a humanoid model. They use this on multiple characters in a shared environment, with the low-level policies shared between them. Both policies are trained using PPO, with the low-level policy learning to match motions from a database of stepping actions using a PD controller.

Hüttenrauch et al. [HSN19] introduce a method that, while not directly applied in crowd simulation, is nevertheless very relevant. Their work focuses on swarm systems, which are inherently similar to crowd scenarios – they both focus on a large number of agents acting in a shared environment, often with a shared goal, with the individual agents typically being indistinguishable. They introduce a method called Mean Feature Embedding, similar to existing Relation Networks [SRB*17, ZRS*18]. This approach uses a modified neural network architecture that ensures permutational invariability between identities of different neighboring agents that are perceived by another agent. This inductive bias can accelerate the training process, and improves scaling to different numbers of agents in the environment. Alonso et al. [APGR20] explore the applicability of RL methods for the task of crowd navigation in AAA games. They use a large and complex 3D environment built on Unity, modelled after real games that typically use a Navigaton Mesh (NavMesh) [Sno00] approach. They use a recurrent LSTM network to give their agents memory, and use the SAC algorithm for policy optimization. As inputs, the agents receive their absolute positions, relative goal position, their speed and acceleration, as well as 3D occupancy maps obtained via box casts, and depth maps from ray casts. They show that this approach has a high level of success, and can enable more flexible map designs, without requiring the designers to specify each possible link.

Zou et al. [ZSSZ18] consider the problem of understanding and predicting crowd behavior specifically from the perspective of Imitation Learning. They introduce a new framework named Social-Aware Generative Adversarial Imitation Learning (SA-GAIL) which is trained to replicate behavior recorded in demonstrations, while disentangling the different factors of decision-making in pedestrian movement. This allows them to obtain a human-understandable interpretation for the model's predictions, as well as for the real data. They use the TRPO algorithm for policy optimization and show that this approach can produce high-quality, interpretable behaviors. A different approach for using data to obtain more human-like behaviors is used by Xu and Karamouzas [XK21]. They use the concept of Knowledge Distillation introduced earlier by Hinton et al. [HVD15]. Along with the standard PPO algorithm for policy optimization, they train a neural

network in a simple supervised way, mapping observations to actions based on data from a crowd motion dataset. Then, the outputs of this network are used as an additional source of reward for the policy learning, encouraging the agent to act similarly to what the supervised network predicts. This way, they obtained more human-like behaviors on typical crowd simulation scenarios, as compared to a regular RL baseline, without a detriment to their performance.

We summarize the algorithms from the papers listed in Sections 6 and 7 in Table 2. While there is a decent diversity of physics engines, as well as a split between TensorFlow in PyTorch for the neural network optimization, the RL algorithm of choice is predominantly PPO.

8. Human Interaction

While not directly a part of Character Animation, interaction between humans and learning-based agents are highly relevant to its applications, notably for Virtual Reality games. For this reason, in this section we describe some of the work towards interactive RL agents. For agents to be interactive, it means that they must be capable of acting in a shared environment with a human-controlled agent, in such a way that they retain their performance on their original goals, while simultaneously reacting to the human's actions appropriately. This is far from trivial, especially when the human behavior significantly differs from the behavior of the trained agents.

An important concept to discuss is the **Theory of Mind (ToM)** [PW78]. Stemming from developmental psychology, ToM refers to the ability that humans and some other animals possess, of reasoning about the internal state of someone else – their goals and beliefs. As Rabinowitz et al. [RPS*18] show, it is also possible to train RL agents so that they can learn ToM of other agents in their environment by observing their actions.

Chodhury et al. [CSHMD19] consider whether it is worthwhile for an agent to learn a full environment model, to learn a ToM model of the human, as opposed to using a model-free approach, in order to cooperate with a human agent. They use an autonomous driving task and show that general black-box model-based methods can work as well as ToM learning, and both of them outperformed the model-free approach.

Carroll et al. [CSH*19] analyze this problem in the general case of cooperative multiagent reinforcement learning, using an environment based on the game Overcook, which requires a high level of cooperation. They find that agents trained in the usual ways, such as with self-play or population-based training, perform significantly when paired with a human played, as compared to their original group performance. They introduce a method based on training a Behavior Cloning agent on data collected from human gameplay. The policy of this agent is then frozen, and it is used as part of the environment dynamics for the actual agent we want to train. Despite the BC agent's low quality, this turns out to be sufficient to improve the performance of the actual agent when evaluated together with a human player.

Christiano et al. [CLB*17] include humans in the loop in the training phase, as opposed to enabling cooperation with them. They

Figure 5: *The relative popularity of PyTorch (red) and TensorFlow (blue) in terms of search volume on Google according to Google Trends, worldwide, between 01/01/2016 and 27/07/2021.*

introduce a method with which it is not necessary to specify a reward function for an agent to optimize. Instead, the algorithm produces demonstrations which are then judged by the human, allowing it to assign reward values from which it learns. This way, RL agents can learn complex behaviors which are not trivial to define mathematically, instead relying on human preferences, while only requiring human input on about 1% of the actual frames used during training.

9. Frameworks

In this section, we discuss the most relevant libraries and frameworks used for training RL agents, which are then applied to character animation. Because the field of Reinforcement Learning relies on neural networks, we begin by describing main frameworks used in Deep Learning. Those are responsible for efficiently performing algebraic operations on tensors (here understood as n-dimensional arrays), using parallelism when possible. They also take care of computing the gradients of functions with the backpropagation algorithm, enabling gradient-based optimization. They do that either on CPU or GPU, sometimes also supporting TPU (Tensor Processing Unit). Then we move on to libraries that function as backbones for RL tasks, by providing standard implementations, offering a common API, or even enabling development of new environments. Finally, we describe the tools used specifically for Reinforcement Learning, either by providing full algorithms, components of them, or other auxiliary functionalities.

All the listed frameworks are written for the Python programming language, although they frequently use other languages for efficient computation that the end user does not need to know. This is the de facto standard in Machine Learning and Reinforcement Learning specifically. Despite its relatively slow performance, through the use of the aforementioned libraries, all the heavy computation is off-loaded to a more efficient language that the end used does not need to use directly.

9.1. Neural Networks

While many open-source tensor computation libraries exist today, three in particular stand out. The first is **TensorFlow (TF)** [MAP*15] developed by Google – originally released in 2015, it underwent major changes in 2019 when the version 2.0 was released with a new philosophy. For this reason, TF1 and TF2 are mutually incompatible and can be seen as two distinct frameworks. The main difference between them is the default handling of computation graphs. In TF1, we have to explicitly define a graph,

Table 2: *A summary of the DRL algorithms, simulation engines, and neural network frameworks in the described papers, where applicable and stated in the paper or the provided source code.* [1] *Value Iteration,* [2] *Open Dynamics Engine,* [3] *Temporal Difference learning,* [4] *Maximum A Posteriori Policy Optimization.*

Citation	Year	Algorithm	Physics Simulation engine	NN Framework
[PBvdP15]	2015	VI[1]	Box2D	–
[PBvdP16]	2016	MACE	Bullet	Caffe
[LH17]	2017	DQN	ODE[2]	Theano
[PBYVDP17]	2017	TD[3]	–	–
[PvdP17]	2017	TD	–	–
[KHL17]	2017	MACE	DART	Caffe
[LH18]	2018	DDPG	ODE	Theano
[PALvdP18]	2018	PPO	Bullet	TensorFlow
[PKM*18]	2018	PPO	Bullet	TensorFlow
[CMM*18]	2018	PPO	MuJoCo	TensorFlow
[YTL18]	2018	PPO	DART	TensorFlow
[CYT*18]	2018	TRPO	DART	PyTorch
[LFL*18]	2018	IPPO	Stage	TensorFlow
[LWL18]	2018	IDDPG	–	TensorFlow
[ZSSZ18]	2018	TRPO + GAIL	–	TensorFlow
[PRL*19]	2019	PPO	DART	TensorFlow
[BCHF19]	2019	PPO	Bullet	TensorFlow
[LPLL19]	2019	PPO	DART	PyTorch
[WL19]	2019	PPO	DART	TensorFlow
[ALX*19]	2019	PPO	Bullet, MuJoCo	PyTorch
[SZQ19]	2019	IPPO + RVO	Unity3D	–
[HSN19]	2019	ITRPO	–	TensorFlow
[PCZ*20]	2020	PPO	Bullet	TensorFlow
[WGSF20]	2020	PPO	Flex	-
[WGH20]	2020	PPO	Bullet	TensorFlow
[YK20]	2020	PPO	MuJoCo	PyTorch
[MTA*20]	2020	V-MPO[4]	MuJoCo	NumPy
[LSCC20]	2020	PPO	Bullet	TensorFlow
[LYZ*20]	2020	PPO	Bullet	PyTorch
[IYOK20]	2020	PPO	MuJoCo	TensorFlow
[XLKvdP20]	2020	PPO	Bullet	PyTorch
[XHLL20]	2020	IPPO + ORCA	–	PyTorch
[HBM*20]	2020	IPPO, MADDPG	Bullet	Caffe
[APGR20]	2020	SAC	Unity3D	–
[PMA*21]	2021	PPO + GAIL	Bullet	TensorFlow
[LLLL21]	2021	PPO	DART	TensorFlow
[MYT*21]	2021	PPO	Bullet	PyTorch
[YWS*21]	2021	PPO	MuJoCo	PyTorch
[YYVDPY21]	2021	PPO	Bullet	PyTorch
[XK21]	2021	IPPO	-	PyTorch

and then run it within a session. This means that any intermediate values are hidden from the user, leading to a difficult debugging process. On the other hand, TF2 uses eager evaluation, building an implicit computation graph. This way, the developer can access the values of any tensors at any time, also in interactive mode e.g. in a Jupyter Notebook [KRKP*16], without needing to modify the computation graph for this purpose. TensorFlow can run computations on CPU, GPU (via CUDA and ROCm), and TPU.

It is important to mention **Keras** [Co15], originally developed as an independent library, is now integrated as part of TF2. It exposes a higher-level API that allows faster development at the cost of fine-tuned control over the computation graph – this, however, can be regained by including lower-level TF2 code as custom layers. While it is primarily designed for Supervised Learning, it is possible to use it with Reinforcement Learning, particularly when using only some of its features in conjunction with TF2 code.

PyTorch [PGM*19], developed by Facebook, was released in 2016 as a Python version of the existing library Torch, which used the Lua language. Its design is very similar to NumPy [HMvdW*20] and inspired TF2 in that it uses eagerly-

Table 3: *A comparison of algorithm support between various frameworks. Legend: ✓ – algorithm supported by the framework, × – algorithm not supported by the framework. Multiagent refers to the capability of training in multiagent environments, with or without parameter sharing. Note that this is not a complete list of algorithms implemented by each framework, as some of them include many other, less relevant algorithms.*

Algorithm	OpenAI Baselines	Stable Baselines	Stable Baselines 3	RLLib	CleanRL	Dopamine	TF-Agents	Tianshou	ML-Agents
DQN	✓	✓	✓	✓	✓	✓	✓	✓	×
Rainbow	×	×	×	✓	×	✓	×	×	×
DDPG	✓	✓	✓	✓	✓	×	✓	✓	×
TD3	×	✓	✓	✓	✓	×	✓	✓	×
SAC	×	✓	✓	✓	✓	×	✓	✓	✓
TRPO	✓	✓	×	×	×	×	×	✓	×
PPO	✓	✓	✓	✓	✓	×	✓	✓	✓
QMIX	×	×	×	✓	×	×	×	×	×
BC/GAIL	✓	✓	×	✓	×	×	×	✓	✓
Multiagent	×	×	×	✓	×	×	×	✓	✓

executed tensors that can be used in dynamic computation graphs. PyTorch can run computation on CPU and GPU (via CUDA), and with some extra effort, TPU. Its simplicity of use, combined with performance that matches TensorFlow, led to its widespread usage, particularly in research context [He19].

A relatively new framework that is worth mentioning is **Jax** [BFH*18]. It offers simple acceleration and parallelization of code with a simple interface that can nearly be used as a drop-in replacement for NumPy. It is heavily inspired by the functional programming paradigm, focuses on composable transformations of functions, notably including differentiating arbitrary functions. By itself, Jax contains efficient numerical operations on tensors, and does not explicitly include neural networks. However, libraries in its ecosystem fill that gap, notably Flax [HLO*20] and Haiku [HCNB20].

In the words of Andrej Karpathy, "I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved." [Kar17] This sentiment is also visible in the data. According to the 2020 Stack Overflow Developer Survey [Sta20], while TensorFlow is more commonly used than PyTorch (11.5% vs 4.6%), it has a lower Loved score (65.2% vs 70.5%), and higher Dreaded score (34.8% vs 29.5%). According to Google Trends, despite its later release date, PyTorch matches the popularity of TensorFlow is terms of search volume, as we show on Figure 5. Overall, the common perception is that TensorFlow is more suited for deployment and industry applications, while PyTorch can be more effective for research and development.

9.2. Environments

The main framework underlying nearly all modern RL research is **Gym** [BCP*16]. It contains a set of commonly used environments that serve as benchmarks for RL algorithms, together with a unified Environment API and a way to implement new environments as Python classes. This way, a single implementation of the environment can be used with multiple algorithms for easier comparison and benchmarking.

The main parts of the Gym API are the following methods:

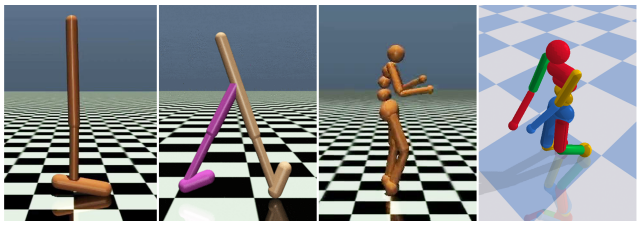

Figure 6: *A visualization of different legged models of varying complexity. The agents' objective is moving each of the joints so that the overall center of mass moves forward or is balanced, while minimizing the energy expenditure. From left to right: Hopper, Walker2d, Humanoid (MuJoCo) and Humanoid (PyBullet).*

- reset() → observation
- step(action) → (observation, reward, done, info)

where the observation and action can be in any format agreed between the environment and the agent. The observations are usually tensors, and actions are either vectors or discrete values, but more complex, tree-like structures are also sometimes used. The reward is a single scalar value, and done is a binary flag indicating whether an episode has ended. Finally, info is a dictionary containing arbitrary additional information. Other properties like observation_space and action_space exist to specify the structure of the information exchanged with the agent, but the extent to which they are required depends on the specific algorithm implementation.

Canonically, Gym only supports partially observable single-agent scenarios, corresponding to POMDPs. Supporting multiagent environments is possible to an extent without changing the abstractions. Specifically, a DecPOMDP can be represented by taking the state and action spaces to be the product spaces of all agents. The reward can then remain as a single scalar value, shared between the agents, maintaining full compatibility with the Gym API.

The situation is more complicated when dealing with more general multiagent problems like POSGs. In this case, each agent may receive its own reward independently of others, which cannot be

represented with a single scalar value. What is more, some agents might not be active throughout the episode. For that reason, frameworks like **RLlib** [LLN*18] use a modified version of a gym Environment where everything is based on Python dictionaries – observations, actions, rewards and 'done' values (i.e. boolean episode termination flags) are Python dictionaries, where each agent's respective values are indexed by the name of that agent.

Other general multiagent formalisms also have their corresponding libraries. The **Petting Zoo** [TBJ*21] library implements the AEC formalism, and is well-suited for general multi-agent problems in which agents do not act simultaneously, but also supports simultaneous actions. Similarly to Gym, it also contains a number of standard benchmark multiagent environments. The EFG formalism is implemented in the OpenSpiel [LLL*20] library, which also contains several ready board and card games.

MuJoCo [TET12b], which stands for Multi-Joint dynamics with Contact, is a physics simulation engine that is widely used in RL research. It can be used to design various robots whose control is then learned with RL algorithms. Certain robots are included in Gym and serve as a common benchmark for new algorithms, notably the Humanoid [TET12a] which is relevant to the topic of character animation. It is worth noting that MuJoCo used to require a paid license to use, but has now become open-source. **PyBullet** [CB16] and **DART** [LGH*18] are common open-source alternatives, filling the same role as a physics simulation engine, and containing many of the same environments ready to use. **NVIDIA Isaac Gym** [LMH*20] fulfills a similar role, enabling very fast parallel simulations accelerated with GPUs. Different legged models from MuJoCo and PyBullet can be seen in Figure 6.

ML-Agents [JBT*20] is a plugin to the Unity game engine which exposes an API through which Python-based agents can interact with games developed in Unity. This can greatly accelerate the development of new environments, as many features from game development are available out-of-the-box, and are relevant for RL tasks. ML-Agents also contains implementations of PPO and SAC algorithms in PyTorch, with the possibility to record and train with user-made demonstrations, using BC or GAIL.

Osim [KMO*18b], based on the OpenSim framework [SSRD11], was used in the 2017 NeurIPS "Learning to Run" challenge is a simulator with a physiologically accurate model of the human body. Its main goal is bridging the gap between biomechanics, neuroscience and computer science communities by providing a common ground for research. It contains a physics simulator, an RL environment, as well as a competition platform to compare different solutions.

Alternatively to gradient-free physical engines, further implementations such as **Nimble** [WOL*21] introduce novel ways of fast and complete differentiable rigid bodies simulations, which can be used for hard optimization problems dealing with complex contact geometries or elastic collisions. Differentiable physics [Mac21] and the possible combination with stochastic gradient-free methods show promising directions for research into more efficient physics engines for learning and optimization.

9.3. Algorithm implementations

In recent years, many frameworks with implementations of RL algorithms have appeared, many of them including the same algorithms, but with differences in the tricks included, or in the implementation philosophy. A common issue is that a framework is too rigid, and therefore it is difficult to customize it for arbitrary research purposes. Here we describe the most relevant frameworks, with a comparison of the algorithms they feature in Table 3

One of the earliest libraries with high-quality implementations of modern RL algorithms is **OpenAI Baselines** [DHK*17]. Released in 2017, it contains the implementations of the most important algorithms existing at the time, including DQN, DDPG and PPO, using TensorFlow. However, at the time of writing it is in maintenance mode, which means that it is not updated with the new developments in the field. Furthermore, the implementations are considered to not be very readable and are challenging to modify or update.

For this reason, the **Stable Baselines (SB2)** [HRE*18] library was developed, with re-implementations of the same algorithms with an addition of SAC and TD3, with a series of improvements to their usability, including: better documentation, tests, using custom policies, and a shared interface between all algorithms. A full comparison is included in the official GitHub repository. Stable Baselines uses TensorFlow for its algorithms, but is now in maintenance mode. The more up-to-date version is **Stable Baselines 3 (SB3)** [RHE*19] which has the same purpose as SB2, but uses PyTorch instead, and is actively updated with new features. Both SB2 and SB3 have very limited support for multiagent training.

RLlib [LLN*18] is built on Ray [MNW*18], a platform for parallel computing, and because of that it can achieve high performance through efficiently parallelizing data collection. It has a large selection of both single-agent and multiagent algorithms in both PyTorch and TensorFlow (depending on the algorithm), with a wide variety of options to adjust for those algorithms. The implementations are very efficient, but the code-base is complex, and therefore difficult to understand and modify for most users.

CleanRL [HDY20] is a library with a different design philosophy – all implementations are contained entirely in a single file. This allows for simple customization, and can serve as a reference when re-implementing those algorithms. CleanRL uses PyTorch for its algorithm implementations. It also includes the Open RL Benchmark, which contains reproducible experiments that use the implemented algorithms on a wide range of standard RL environments.

Dopamine [CMG*18] is a research framework developed by Google, with the design principles of *easy experimentation*, *flexible development*, *compactness and reliability*, and *reproducibility*. It focuses on variants of DQN, including Rainbow and several other modifications. The framework primarily uses TensorFlow, but it also contains Jax implementations of the algorithms.

TF-Agents [GKR*18] is a part of the TensorFlow ecosystem dedicated to Bandits and Reinforcement Learning algorithms. It contains high-quality implementations of modern RL algorithms written in TF2. The implementations are modular and include tests, benchmarks and tutorials on how to use the library.

Tianshou [WCY*21] maintained by researchers from Tsinghua

University is a modular RL framework based on PyTorch. While focusing on single-agent model-free algorithms, it also supports multiagent environments, model-based algorithms and imitation learning. The implementations are flexible so that it is possible to modify them for research purposes, and very efficient due to parallelization.

Rlax [BHQ*20], developed by Deepmind as part of the Jax ecosystem, takes a different approach than all the aforementioned frameworks. Instead of full algorithms, it contains building blocks that can then be used in implementing the algorithms. This includes exploration strategies, policies, update strategies and more.

9.4. Summary

As we show, there are numerous libraries that can be used for creating environments and training RL algorithms, and the best choice will necessarily depend on the application. Regarding environment creation, highly specialized problems might require custom simulators, but for generic character and crowd animation problems, we recommend using the Unity engine with ML-Agents to build a gym or gym-like interface. For training, if customization of the algorithm is unnecessary, we recommend using either RLlib or Stable Baselines 3 for single-agent (skeletal animation) scenarios. If some degree of customization is necessary, Stable Baselines 3 or Tianshou are worthwhile options, with RLlib typically being too rigid. Finally, if it is necessary to introduce major changes to the typical reinforcement learning loop, it might be necessary to use a custom-written algorithm - components from Tianshou, CleanRL and Rlax can then prove to be helpful. Naturally, all of this is conditioned on the availability of the desired algorithm in a specific framework (see Table 3).

10. Conclusions

Reinforcement Learning is a rapidly growing field of Artificial Intelligence and Machine Learning, concerned with authoring intelligent behaviors by specifying their tasks, instead of describing the specific behaviors. This method is of high utility for applications in Character Animation, both for individual characters, as well as entire crowds.

In fact, many works already use RL algorithms to create more believable or higher-quality animations. In many cases, with an appropriate simulator, it is sufficient to specify the desired task in terms of a reward function (e.g. agents in a crowd heading towards a certain goal, while avoiding collisions with one another), and then train one of the state-of-the-art algorithms to obtain interesting behaviors. However, Deep Reinforcement Learning is still a relatively young field, and thus its use is not common in the industry. This is likely to change in the upcoming years.

As for the implementation, there are many resources available to significantly accelerate the development of new applications of RL. While different frameworks excel in different aspects, there exist options for diverse use cases and degrees of complexity, so that in-depth expertise in the inner workings of RL algorithms is not absolutely necessary to be able to apply them.

We anticipate significant progress on the intersection of Character Animation and Reinforcement Learning in the upcoming years. The algorithms become more and more efficient, seeing success after success in classic challenges like the games of Go and Starcraft. This, combined with the widespread usage of GPUs, will make it possible to seamlessly integrate them into typical Computer Graphics workflows.

Acknowledgement

This work has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 860768 (CLIPE project).

References

[Ach18] ACHIAM J.: Spinning up in deep reinforcement learning. 4

[AF02] ARIKAN O., FORSYTH D. A.: Interactive motion generation from examples. *ACM Transactions on Graphics (TOG) 21*, 3 (2002), 483–490. 13

[Ale20] ALEXANDER S. A.: The archimedean trap: Why traditional reinforcement learning will probably not yield AGI. 70–85. 6

[ALX*19] ABDOLHOSSEINI F., LING H. Y., XIE Z., PENG X. B., VAN DE PANNE M.: On learning symmetric locomotion. In *Motion, Interaction and Games* (2019), MIG '19, Association for Computing Machinery. event-place: Newcastle upon Tyne, United Kingdom. 16, 19

[AMC20] AMIT R., MEIR R., CIOSEK K.: Discount factor as a regularizer in reinforcement learning. In *International Conference on Machine Learning* (2020), PMLR, pp. 269–278. ISSN: 2640-3498. 6

[APGR20] ALONSO E., PETER M., GOUMARD D., ROMOFF J.: Deep reinforcement learning for navigation in AAA video games. 17, 19

[ARS*20] ANDRYCHOWICZ M., RAICHUK A., STAŃCZYK P., ORSINI M., GIRGIN S., MARINIER R., HUSSENOT L., GEIST M., PIETQUIN O., MICHALSKI M., GELLY S., BACHEM O.: What matters in on-policy reinforcement learning? a large-scale empirical study. 10

[BCHF19] BERGAMIN K., CLAVET S., HOLDEN D., FORBES J. R.: Drecon: data-driven responsive control of physics-based characters. *ACM Transactions On Graphics (TOG) 38*, 6 (2019), 1–11. 15, 19

[BCP*16] BROCKMAN G., CHEUNG V., PETTERSSON L., SCHNEIDER J., SCHULMAN J., TANG J., ZAREMBA W.: OpenAI gym. 14, 20

[BDM17] BELLEMARE M. G., DABNEY W., MUNOS R.: A distributional perspective on reinforcement learning. In *International Conference on Machine Learning* (2017), PMLR, pp. 449–458. ISSN: 2640-3498. 8

[Bel57] BELLMAN R.: A markovian decision process. 679–684. Publisher: Indiana University Mathematics Department. 2

[Bel03] BELLMAN R. E.: *Dynamic Programming*. Dover Publications, Inc., 2003. 5

[BFH*18] BRADBURY J., FROSTIG R., HAWKINS P., JOHNSON M. J., LEARY C., MACLAURIN D., WANDERMAN-MILNE S.: JAX: composable transformations of python+NumPy programs, 2018. 20

[BHQ*20] BUDDEN D., HESSEL M., QUAN J., KAPTUROWSKI S., BAUMLI K., BHUPATIRAJU S., GUY A., KING M.: RLax: Reinforcement learning in JAX, 2020. 22

[BS99] BAIN M., SAMMUT C.: A framework for behavioural cloning. In *Machine Intelligence 15, Intelligent Agents [St. Catherine's College, Oxford, July 1995]* (1999), Oxford University, pp. 103–129. 12

[BZI00] BERNSTEIN D. S., ZILBERSTEIN S., IMMERMAN N.: The complexity of decentralized control of markov decision processes. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence* (2000), UAI'00, Morgan Kaufmann Publishers Inc., pp. 32–37. 3

[CB16] COUMANS E., BAI Y.: *PyBullet, a Python module for physics simulation for games, robotics and machine learning*. 2016. 14, 21

[CBVdP09] COROS S., BEAUDOIN P., VAN DE PANNE M.: Robust task-based control policies for physics-based characters. In *ACM SIGGRAPH Asia 2009 papers*. 2009, pp. 1–9. 15

[CLB*17] CHRISTIANO P. F., LEIKE J., BROWN T. B., MARTIC M., LEGG S., AMODEI D.: Deep reinforcement learning from human preferences. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017), NIPS'17, Curran Associates Inc., pp. 4302–4310. event-place: Long Beach, California, USA. 18

[CMG*18] CASTRO P. S., MOITRA S., GELADA C., KUMAR S., BELLEMARE M. G.: Dopamine: A research framework for deep reinforcement learning. 21

[CMM*18] CHENTANEZ N., MÜLLER M., MACKLIN M., MAKOVIYCHUK V., JESCHKE S.: Physics-based motion capture imitation with deep reinforcement learning. In *Proceedings of the 11th annual international conference on motion, interaction, and games* (2018), pp. 1–10. 15, 19

[Co15] CHOLLET F., OTHERS: Keras, 2015. 19

[Cou06] COULOM R.: Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th international conference on Computers and games* (2006), CG'06, Springer-Verlag, pp. 72–83. 7

[CSH*19] CARROLL M., SHAH R., HO M. K., GRIFFITHS T. L., SESHIA S. A., ABBEEL P., DRAGAN A.: On the utility of learning about humans for human-AI coordination. 18

[CSHMD19] CHOUDHURY R., SWAMY G., HADFIELD-MENELL D., DRAGAN A. D.: On the utility of model learning in HRI. In *Proceedings of the 14th ACM/IEEE International Conference on Human-Robot Interaction* (2019), HRI '19, IEEE Press, pp. 317–325. event-place: Daegu, Republic of Korea. 18

[CYT*18] CLEGG A., YU W., TAN J., LIU C. K., TURK G.: Learning to dress: Synthesizing human dressing motion via deep reinforcement learning. Place: New York, NY, USA Publisher: Association for Computing Machinery. 16, 19

[DBH16] DAFTRY S., BAGNELL J. A., HEBERT M.: Learning transferable policies for monocular reactive MAV control. _eprint: 1608.00627. 12

[DHK*17] DHARIWAL P., HESSE C., KLIMOV O., NICHOL A., PLAPPERT M., RADFORD A., SCHULMAN J., SIDOR S., WU Y., ZHOKHOV P.: OpenAI baselines, 2017. Publication Title: GitHub repository. 21

[EIS*20] ENGSTROM L., ILYAS A., SANTURKAR S., TSIPRAS D., JANOOS F., RUDOLPH L., MADRY A.: Implementation matters in deep policy gradients: A case study on PPO and TRPO. 10

[FAP*17] FORTUNATO M., AZAR M. G., PIOT B., MENICK J., OSBAND I., GRAVES A., MNIH V., MUNOS R., HASSABIS D., PIETQUIN O., BLUNDELL C., LEGG S.: Noisy networks for exploration. 8

[FGB*19] FEDUS W., GELADA C., BENGIO Y., BELLEMARE M. G., LAROCHELLE H.: Hyperbolic discounting and learning over multiple horizons. 6

[FvHM18] FUJIMOTO S., VAN HOOF H., MEGER D.: Addressing function approximation error in actor-critic methods. 11

[GBC16] GOODFELLOW I., BENGIO Y., COURVILLE A.: *Deep Learning*. MIT Press, 2016. 4

[GEK17] GUPTA J. K., EGOROV M., KOCHENDERFER M.: Cooperative multi-agent control using deep reinforcement learning. In *Autonomous Agents and Multiagent Systems* (2017), Sukthankar G., Rodriguez-Aguilar J. A., (Eds.), Springer International Publishing, pp. 66–83. 12

[GGL*20] GOECKS V. G., GREMILLION G. M., LAWHERN V. J., VALASEK J., WAYTOWICH N. R.: Integrating behavior cloning and reinforcement learning for improved performance in dense and sparse reward environments. In *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems* (2020), AAMAS '20, International Foundation for Autonomous Agents and Multiagent Systems, pp. 465–473. event-place: Auckland, New Zealand. 12

[GKR*18] GUADARRAMA S., KORATTIKARA A., RAMIREZ O., CASTRO P., HOLLY E., FISHMAN S., WANG K., GONINA E., WU N., KOKIOPOULOU E., SBAIZ L., SMITH J., BARTÓK G., BERENT J., HARRIS C., VANHOUCKE V., BREVDO E.: TF-agents: A library for reinforcement learning in TensorFlow, 2018. 21

[GPAM*20] GOODFELLOW I., POUGET-ABADIE J., MIRZA M., XU B., WARDE-FARLEY D., OZAIR S., COURVILLE A., BENGIO Y.: Generative adversarial networks. 139–144. Place: New York, NY, USA Publisher: Association for Computing Machinery. 12

[GTZL20] GUO D., TANG L., ZHANG X., LIANG Y.-C.: Joint optimization of handover control and power allocation based on multi-agent deep reinforcement learning. 13124–13138. Conference Name: IEEE Transactions on Vehicular Technology. 12

[HBM*20] HAWORTH B., BERSETH G., MOON S., FALOUTSOS P., KAPADIA M.: Deep integration of physical humanoid control and crowd navigation. pp. 1–10. 17, 19

[HBZ04] HANSEN E. A., BERNSTEIN D. S., ZILBERSTEIN S.: Dynamic programming for partially observable stochastic games. In *Proceedings of the 19th national conference on Artifical intelligence* (2004), AAAI'04, AAAI Press, pp. 709–715. 3

[HCNB20] HENNIGAN T., CAI T., NORMAN T., BABUSCHKIN I.: Haiku: Sonnet for JAX, 2020. 20

[HDL16] HA D., DAI A., LE Q. V.: HyperNetworks. 13

[HDY20] HUANG S., DOSSA R., YE C.: CleanRL: High-quality single-file implementation of deep reinforcement learning algorithms, 2020. Publication Title: GitHub repository. 21

[HE16] HO J., ERMON S.: Generative adversarial imitation learning. 12

[He19] HE H.: The state of machine learning frameworks in 2019. 20

[HET*14] HÄMÄLÄINEN P., ERIKSSON S., TANSKANEN E., KYRKI V., LEHTINEN J.: Online motion synthesis using sequential monte carlo. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 1–12. 15

[HHL21] HU S., HU J., LIAO S.-W.: Noisy-MAPPO: Noisy credit assignment for cooperative multi-agent actor-critic methods. 12

[HIB*17] HENDERSON P., ISLAM R., BACHMAN P., PINEAU J., PRECUP D., MEGER D.: Deep reinforcement learning that matters. 10

[HLO*20] HEEK J., LEVSKAYA A., OLIVER A., RITTER M., RONDEPIERRE B., STEINER A., ZEE M. V.: Flax: A neural network library and ecosystem for JAX, 2020. 20

[HMvdW*20] HARRIS C. R., MILLMAN K. J., VAN DER WALT S. J., GOMMERS R., VIRTANEN P., COURNAPEAU D., WIESER E., TAYLOR J., BERG S., SMITH N. J., KERN R., PICUS M., HOYER S., VAN KERKWIJK M. H., BRETT M., HALDANE A., DEL RÍO J. F., WIEBE M., PETERSON P., GÉRARD-MARCHANT P., SHEPPARD K., REDDY T., WECKESSER W., ABBASI H., GOHLKE C., OLIPHANT T. E.: Array programming with NumPy. 357–362. Number: 7825 Publisher: Nature Publishing Group. 19

[HMvH*17] HESSEL M., MODAYIL J., VAN HASSELT H., SCHAUL T., OSTROVSKI G., DABNEY W., HORGAN D., PIOT B., AZAR M., SILVER D.: Rainbow: Combining improvements in deep reinforcement learning. 8

[HRE*18] HILL A., RAFFIN A., ERNESTUS M., GLEAVE A., KANERVISTO A., TRAORE R., DHARIWAL P., HESSE C., KLIMOV O., NICHOL A., PLAPPERT M., RADFORD A., SCHULMAN J., SIDOR S., WU Y.: Stable baselines, 2018. Publication Title: GitHub repository. 21

[HRL15] HÄMÄLÄINEN P., RAJAMÄKI J., LIU C. K.: Online control of simulated humanoids using particle belief propagation. *ACM Transactions on Graphics (TOG) 34*, 4 (2015), 1–13. 15

[HS97] HOCHREITER S., SCHMIDHUBER J.: Long short-term memory. 1735–1780. 17

[HSN19] HÜTTENRAUCH M., SOSIC A., NEUMANN G.: Deep reinforcement learning for swarm systems. 17, 19

[HVD15] HINTON G., VINYALS O., DEAN J.: Distilling the knowledge in a neural network. 17

[HWH*21] HU J., WU H., HARDING S. A., JIANG S., LIAO S.-W.: RIIT: Rethinking the importance of implementation tricks in multi-agent reinforcement learning. 13

[HZAL18] HAARNOJA T., ZHOU A., ABBEEL P., LEVINE S.: Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. 11

[IYOK20] ISOGAWA M., YUAN Y., O'TOOLE M., KITANI K.: Optical non-line-of-sight physics-based 3d human pose estimation. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2020), IEEE, pp. 7011–7020. 15, 19

[JBT*20] JULIANI A., BERGES V.-P., TENG E., COHEN A., HARPER J., ELION C., GOY C., GAO Y., HENRY H., MATTAR M., LANGE D.: Unity: A general platform for intelligent agents. 21

[Jon20] JONES A. L.: A clearer proof of the policy gradient theorem. 4

[JVWDGL19] JIANG Y., VAN WOUWE T., DE GROOTE F., LIU C. K.: Synthesis of biologically realistic human motion using joint torque actuation. Place: New York, NY, USA Publisher: Association for Computing Machinery. 16

[Kak01] KAKADE S.: A natural policy gradient. In *Proceedings of the 14th International Conference on Neural Information Processing Systems: Natural and Synthetic* (2001), NIPS'01, MIT Press, pp. 1531–1538. 9

[Kar17] KARPATHY A.: I've been using PyTorch a few months now and i've never felt better. [...], 2017. 20

[KGP02] KOVAR L., GLEICHER M., PIGHIN F.: Motion graphs. *Proceedings of ACM SIGGRAPH 2002, July* (2002), 473–482. 13

[KHL17] KUMAR V. C. V., HA S., LIU C. K.: Learning a unified control policy for safe falling. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (2017), pp. 3940–3947. 16, 19

[KLC98] KAELBLING L. P., LITTMAN M. L., CASSANDRA A. R.: Planning and acting in partially observable stochastic domains. 99–134. 3

[KMO*18a] KIDZIŃSKI L., MOHANTY S. P., ONG C., HICKS J., FRANCIS S., LEVINE S., SALATHÉ M., DELP S.: Learning to run challenge: Synthesizing physiologically accurate motion using deep reinforcement learning. In *NIPS 2017 Competition Book*, Escalera S., Weimer M., (Eds.). Springer, 2018. 16

[KMO*18b] KIDZIŃSKI L., MOHANTY S. P., ONG C., HUANG Z., ZHOU S., PECHENKO A., STELMASZCZYK A., JAROSIK P., PAVLOV M., KOLESNIKOV S., PLIS S., CHEN Z., ZHANG Z., CHEN J., SHI J., ZHENG Z., YUAN C., LIN Z., MICHALEWSKI H., MIŁOŚ P., OSIŃSKI B., MELNIK A., SCHILLING M., RITTER H., CARROLL S., HICKS J., LEVINE S., SALATHÉ M., DELP S.: Learning to run challenge solutions: Adapting reinforcement learning methods for neuromusculoskeletal environments. 21

[KRKP*16] KLUYVER T., RAGAN-KELLEY B., PÉREZ F., GRANGER B., BUSSONNIER M., FREDERIC J., KELLEY K., HAMRICK J., GROUT J., CORLAY S., IVANOV P., AVILA D., ABDALLA S., WILLING C., TEAM J. D.: Jupyter notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (2016), Loizides F., Scmidt B., (Eds.), IOS Press, pp. 87–90. 19

[LCR*02] LEE J., CHAI J., REITSMA P. S., HODGINS J. K., POLLARD N. S.: Interactive control of avatars animated with human motion data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques* (2002), pp. 491–500. 13

[LFL*18] LONG P., FAN T., LIAO X., LIU W., ZHANG H., PAN J.: Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning. 17, 19

[LGH*18] LEE J., GREY M. X., HA S., KUNZ T., JAIN S., YE Y., SRINIVASA S. S., STILMAN M., LIU C. K.: Dart: Dynamic animation and robotics toolkit. *The Journal of Open Source Software 3*, 22 (2018). 21

[LH11] LATTIMORE T., HUTTER M.: Time consistent discounting. In *Algorithmic Learning Theory* (2011), Kivinen J., Szepesvári C., Ukkonen E., Zeugmann T., (Eds.), Lecture Notes in Computer Science, Springer, pp. 383–397. 6

[LH17] LIU L., HODGINS J.: Learning to schedule control fragments for physics-based characters using deep q-learning. *ACM Transactions on Graphics (TOG) 36*, 3 (2017), 1–14. 14, 19

[LH18] LIU L., HODGINS J.: Learning basketball dribbling skills using trajectory optimization and deep reinforcement learning. *ACM Transactions on Graphics (TOG) 37*, 4 (2018), 1–14. 14, 19

[LHP*15] LILLICRAP T. P., HUNT J. J., PRITZEL A., HEESS N., EREZ T., TASSA Y., SILVER D., WIERSTRA D.: Continuous control with deep reinforcement learning. 6, 10

[LL06] LEE J., LEE K. H.: Precomputing avatar behavior from human motion data. *Graphical models 68*, 2 (2006), 158–174. 13

[LLL*20] LANCTOT M., LOCKHART E., LESPIAU J.-B., ZAMBALDI V., UPADHYAY S., PÉROLAT J., SRINIVASAN S., TIMBERS F., TUYLS K., OMIDSHAFIEI S., HENNES D., MORRILL D., MULLER P., EWALDS T., FAULKNER R., KRAMÁR J., DE VYLDER B., SAETA B., BRADBURY J., DING D., BORGEAUD S., LAI M., SCHRITTWIESER J., ANTHONY T., HUGHES E., DANIHELKA I., RYAN-DAVIS J.: OpenSpiel: A framework for reinforcement learning in games. 3, 21

[LLLL21] LEE S., LEE S., LEE Y., LEE J.: Learning a family of motor skills from a single motion clip. *ACM Transactions on Graphics (TOG) 40*, 4 (2021), 1–13. 15, 19

[LLN*18] LIANG E., LIAW R., NISHIHARA R., MORITZ P., FOX R., GOLDBERG K., GONZALEZ J., JORDAN M., STOICA I.: RLlib: Abstractions for distributed reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning* (2018), Dy J., Krause A., (Eds.), vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 3053–3062. 21

[LMH*20] LIANG J., MAKOVIYCHUK V., HANDA A., CHENTANEZ N., MACKLIN M., FOX D.: GPU-accelerated robotic simulation for distributed reinforcement learning. 21

[LPLL19] LEE S., PARK M., LEE K., LEE J.: Scalable muscle-actuated human simulation and control. *ACM Transactions On Graphics (TOG) 38*, 4 (2019), 1–13. 15, 19

[LPY16] LIU L., PANNE M. V. D., YIN K.: Guided learning of control graphs for physics-based characters. *ACM Transactions on Graphics (TOG) 35*, 3 (2016), 1–14. 14

[LSCC20] LUO Y.-S., SOESENO J. H., CHEN T. P.-C., CHEN W.-C.: CARL: Controllable agent with reinforcement learning for quadruped locomotion. Place: New York, NY, USA Publisher: Association for Computing Machinery. 15, 16, 19

[LWB*10] LEE Y., WAMPLER K., BERNSTEIN G., POPOVIĆ J., POPOVIĆ Z.: Motion fields for interactive character locomotion. 138:1–138:8. 13

[LWL18] LEE J., WON J., LEE J.: Crowd simulation by deep reinforcement learning. In *Proceedings of the 11th Annual International Conference on Motion, Interaction, and Games* (2018), ACM, pp. 1–7. 17, 19

[LWL*20] LIU D., WANG Z., LU B., CONG M., YU H., ZOU Q.: A reinforcement learning-based framework for robot manipulation skill acquisition. 108429–108437. Conference Name: IEEE Access. 12

[LWT*20] LOWE R., WU Y., TAMAR A., HARB J., ABBEEL P., MORDATCH I.: Multi-agent actor-critic for mixed cooperative-competitive environments. 12

[LYvdP*10] LIU L., YIN K., VAN DE PANNE M., SHAO T., XU W.: Sampling-based contact-rich motion control. In *ACM SIGGRAPH 2010 papers*. 2010, pp. 1–10. 14

[LYZ*20] LING H. Y., YU LING H., ZINNO F., CHENG G., VAN DE PANNE M.: Character Controllers Using Motion VAEs. *ACM Trans. Graph 39*, 4 (2020), 12. 13, 19

[Mac21] MACKLIN M.: Differentiable Physics Simulation for Learning and Robotics. In *GTC 2021* (2021). 21

[MAP*15] MARTÍN ABADI, ASHISH AGARWAL, PAUL BARHAM, EUGENE BREVDO, ZHIFENG CHEN, CRAIG CITRO, GREG S. CORRADO, ANDY DAVIS, JEFFREY DEAN, MATTHIEU DEVIN, SANJAY GHEMAWAT, IAN GOODFELLOW, ANDREW HARP, GEOFFREY IRVING, MICHAEL ISARD, JIA Y., RAFAL JOZEFOWICZ, LUKASZ KAISER, MANJUNATH KUDLUR, JOSH LEVENBERG, DAN MANÉ, RAJAT MONGA, SHERRY MOORE, DEREK MURRAY, CHRIS OLAH, MIKE SCHUSTER, JONATHON SHLENS, BENOIT STEINER, ILYA SUTSKEVER, KUNAL TALWAR, PAUL TUCKER, VINCENT VANHOUCKE, VIJAY VASUDEVAN, FERNANDA VIÉGAS, ORIOL VINYALS, PETE WARDEN, MARTIN WATTENBERG, MARTIN WICKE, YUAN YU, XIAOQIANG ZHENG: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. 18

[MBM*16] MNIH V., BADIA A. P., MIRZA M., GRAVES A., HARLEY T., LILLICRAP T. P., SILVER D., KAVUKCUOGLU K.: Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48* (2016), ICML'16, JMLR.org, pp. 1928–1937. 10

[MHC*21] MOUROT L., HOYET L., CLERC F. L., SCHNITZLER F., HELLIER P.: A survey on deep learning for skeleton-based human animation. *arXiv preprint arXiv:2110.06901* (2021). 2

[MKS*15] MNIH V., KAVUKCUOGLU K., SILVER D., RUSU A. A., VENESS J., BELLEMARE M. G., GRAVES A., RIEDMILLER M., FIDJELAND A. K., OSTROVSKI G., PETERSEN S., BEATTIE C., SADIK A., ANTONOGLOU I., KING H., KUMARAN D., WIERSTRA D., LEGG S., HASSABIS D.: Human-level control through deep reinforcement learning. 529–533. 6, 8

[MNW*18] MORITZ P., NISHIHARA R., WANG S., TUMANOV A., LIAW R., LIANG E., ELIBOL M., YANG Z., PAUL W., JORDAN M. I., STOICA I.: Ray: A distributed framework for emerging AI applications. 21

[MTA*20] MEREL J., TUNYASUVUNAKOOL S., AHUJA A., TASSA Y., HASENCLEVER L., PHAM V., EREZ T., WAYNE G., HEESS N.: Catch & carry: reusable neural controllers for vision-guided whole-body tasks. *ACM Transactions on Graphics (TOG) 39*, 4 (2020), 39–1. 15, 19

[MTP12] MORDATCH I., TODOROV E., POPOVIĆ Z.: Discovery of complex behaviors through contact-invariant optimization. *ACM Transactions on Graphics 31*, 4 (2012). 15

[MYT*21] MA L.-K., YANG Z., TONG X., GUO B., YIN K.: Learning and exploring motor skills with spacetime bounds. In *Computer Graphics Forum* (2021), vol. 40, Wiley Online Library, pp. 251–263. 15, 19

[NHR99] NG A. Y., HARADA D., RUSSELL S. J.: Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning* (1999), ICML '99, Morgan Kaufmann Publishers Inc., pp. 278–287. 4

[PALvdP18] PENG X. B., ABBEEL P., LEVINE S., VAN DE PANNE M.: DeepMimic: Example-guided deep reinforcement learning of physics-based character skills. Place: New York, NY, USA Publisher: Association for Computing Machinery. 14, 15, 16, 19

[PBvdP15] PENG X. B., BERSETH G., VAN DE PANNE M.: Dynamic terrain traversal skills using reinforcement learning. Place: New York, NY, USA Publisher: Association for Computing Machinery. 15, 19

[PBvdP16] PENG X. B., BERSETH G., VAN DE PANNE M.: Terrain-adaptive locomotion skills using deep reinforcement learning. Place: New York, NY, USA Publisher: Association for Computing Machinery. 15, 19

[PBYVDP17] PENG X. B., BERSETH G., YIN K., VAN DE PANNE M.: DeepLoco: Dynamic locomotion skills using hierarchical deep reinforcement learning. Place: New York, NY, USA Publisher: Association for Computing Machinery. 14, 19

[PCZ*20] PENG X. B., COUMANS E., ZHANG T., LEE T.-W., TAN J., LEVINE S.: Learning agile robotic locomotion skills by imitating animals. *arXiv preprint arXiv:2004.00784* (2020). 14, 19

[PGM*19] PASZKE A., GROSS S., MASSA F., LERER A., BRADBURY J., CHANAN G., KILLEEN T., LIN Z., GIMELSHEIN N., ANTIGA L., DESMAISON A., KOPF A., YANG E., DEVITO Z., RAISON M., TEJANI A., CHILAMKURTHY S., STEINER B., FANG L., BAI J., CHINTALA S.: PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, Wallach H., Larochelle H., Beygelzimer A., Alché-Buc F. d., Fox E., Garnett R., (Eds.). Curran Associates, Inc., 2019, pp. 8024–8035. 19

[PKM*18] PENG X. B., KANAZAWA A., MALIK J., ABBEEL P., LEVINE S.: Sfv: Reinforcement learning of physical skills from videos. *ACM Transactions On Graphics (TOG) 37*, 6 (2018), 1–14. 15, 19

[PMA*21] PENG X. B., MA Z., ABBEEL P., LEVINE S., KANAZAWA A.: AMP: Adversarial motion priors for stylized physics-based character control. Place: New York, NY, USA Publisher: Association for Computing Machinery. 14, 15, 19

[PRL*19] PARK S., RYU H., LEE S., LEE S., LEE J.: Learning predict-and-simulate policies from unorganized human motion data. *ACM Transactions on Graphics (TOG) 38*, 6 (2019), 1–11. 14, 15, 19

[PvdP17] PENG X. B., VAN DE PANNE M.: Learning locomotion skills using deeprl: Does the choice of action space matter? In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2017), pp. 1–13. 15, 19

[PW78] PREMACK D., WOODRUFF G.: Does the chimpanzee have a theory of mind? 515–526. 18

[RFPW20] RASHID T., FARQUHAR G., PENG B., WHITESON S.: Weighted QMIX: Expanding monotonic value function factorisation for deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (2020), Larochelle H., Ranzato M., Hadsell R., Balcan M.-F., Lin H.-T., (Eds.). 13

[RGB11] ROSS S., GORDON G., BAGNELL D.: A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics* (2011), Gordon G., Dunson D., Dudík M., (Eds.), vol. 15 of *Proceedings of Machine Learning Research*, PMLR, pp. 627–635. 12

[RH17] RAJAMÄKI J., HÄMÄLÄINEN P.: Augmenting sampling based controllers with machine learning. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (2017), pp. 1–9. 15

[RHE*19] RAFFIN A., HILL A., ERNESTUS M., GLEAVE A., KANERVISTO A., DORMANN N.: *Stable Baselines3*. GitHub, 2019. Publication Title: GitHub repository. 21

[RPS*18] RABINOWITZ N. C., PERBET F., SONG H. F., ZHANG C., ESLAMI S. M. A., BOTVINICK M.: Machine theory of mind. 18

[RSdW*18] RASHID T., SAMVELYAN M., DE WITT C. S., FARQUHAR G., FOERSTER J., WHITESON S.: QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning. 13

[RTvdP20] REDA D., TAO T., VAN DE PANNE M.: Learning to locomote: Understanding how environment design matters for deep reinforcement learning. In *Motion, Interaction and Games*. 2020, pp. 1–10. 14

[Rub81] RUBINSTEIN R. Y.: *Simulation and the Monte Carlo Method*, 1st ed. John Wiley & Sons, Inc., 1981. 5

[SB98] SUTTON R. S., BARTO A. G.: *Introduction to Reinforcement Learning*, 1st ed. MIT Press, 1998. 2

[SB18] SUTTON R. S., BARTO A. G.: *Reinforcement Learning: An Introduction*. A Bradford Book, 2018. 2, 5, 6

[SHS*17] SILVER D., HUBERT T., SCHRITTWIESER J., ANTONOGLOU I., LAI M., GUEZ A., LANCTOT M., SIFRE L., KUMARAN D., GRAEPEL T., LILLICRAP T., SIMONYAN K., HASSABIS D.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. 7

[SKK*19] SON K., KIM D., KANG W. J., HOSTALLERO D. E., YI Y.: QTRAN: Learning to factorize with transformation for cooperative multi-agent reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning* (2019), Chaudhuri K., Salakhutdinov R., (Eds.), vol. 97 of *Proceedings of Machine Learning Research*, PMLR, pp. 5887–5896. 13

[SLA*15] SCHULMAN J., LEVINE S., ABBEEL P., JORDAN M., MORITZ P.: Trust region policy optimization. In *International Conference on Machine Learning* (2015), PMLR, pp. 1889–1897. ISSN: 1938-7228. 9

[SLG*17] SUNEHAG P., LEVER G., GRUSLYS A., CZARNECKI W. M., ZAMBALDI V., JADERBERG M., LANCTOT M., SONNERAT N., LEIBO J. Z., TUYLS K., GRAEPEL T.: Value-decomposition networks for cooperative multi-agent learning. 13

[SLH*14] SILVER D., LEVER G., HEESS N., DEGRIS T., WIERSTRA D., RIEDMILLER M.: Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32* (2014), ICML'14, JMLR.org, pp. I–387–I–395. 10

[SML*18] SCHULMAN J., MORITZ P., LEVINE S., JORDAN M., ABBEEL P.: High-dimensional continuous control using generalized advantage estimation. 10

[SMSM99] SUTTON R. S., MCALLESTER D., SINGH S., MANSOUR Y.: Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems* (1999), NIPS'99, MIT Press, pp. 1057–1063. 4, 6, 9

[Sno00] SNOOK G.: Simplified 3d movement and pathfinding using navigation meshes. In *Game Programming Gems*, DeLoura M., (Ed.). Charles River Media, 2000, pp. 288–304. 17

[Soa18] SOARES N.: The value learning problem. In *Artificial Intelligence Safety and Security*, Yampolskiy R. V., (Ed.), 1 ed. Chapman and Hall/CRC, 2018, pp. 89–97. 6

[SQAS16] SCHAUL T., QUAN J., ANTONOGLOU I., SILVER D.: Prioritized experience replay. 8

[SRB*17] SANTORO A., RAPOSO D., BARRETT D. G. T., MALINOWSKI M., PASCANU R., BATTAGLIA P., LILLICRAP T.: A simple neural network module for relational reasoning. 17

[SSPS21] SILVER D., SINGH S., PRECUP D., SUTTON R. S.: Reward is enough. 103535. 6

[SSRD11] SETH A., SHERMAN M., REINBOLT J. A., DELP S. L.: OpenSim: a musculoskeletal modeling and simulation framework for in silico investigations and exchange. 212–232. 16, 21

[Sta20] STACK OVERFLOW: Stack overflow developer survey 2020, 2020. 20

[Sut88] SUTTON R. S.: Learning to predict by the methods of temporal differences. 9–44. 8, 10

[SWD*17] SCHULMAN J., WOLSKI F., DHARIWAL P., RADFORD A., KLIMOV O.: Proximal policy optimization algorithms. 6, 9

[SZQ19] SUN L., ZHAI J., QIN W.: Crowd navigation in an unknown and dynamic environment based on deep reinforcement learning. 109544–109554. Conference Name: IEEE Access. 17, 19

[TBJ*21] TERRY J. K., BLACK B., JAYAKUMAR M., HARI A., SULLIVAN R., SANTOS L., DIEFFENDAHL C., WILLIAMS N. L., LOKESH Y., HORSCH C., RAVI P.: PettingZoo: Gym for multi-agent reinforcement learning. 3, 21

[TET12a] TASSA Y., EREZ T., TODOROV E.: Synthesis and stabilization of complex behaviors through online trajectory optimization. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), pp. 4906–4913. 15, 21

[TET12b] TODOROV E., EREZ T., TASSA Y.: MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2012), pp. 5026–5033. ISSN: 2153-0866. 14, 21

[TGB*21] TERRY J. K., GRAMMEL N., BLACK B., HARI A., HORSCH C., SANTOS L.: Agent environment cycle games. 3

[TGH*20] TERRY J. K., GRAMMEL N., HARI A., SANTOS L., BLACK B.: Revisiting parameter sharing in multi-agent deep reinforcement learning. 12

[TLP07] TREUILLE A., LEE Y., POPOVIĆ Z.: Near-optimal character animation with continuous control. In *ACM SIGGRAPH 2007 papers*. 2007, pp. 7–es. 13

[TP21] TOLL W., PETTRÉ J.: Algorithms for microscopic crowd simulation: Advancements in the 2010s. 731–754. 2

[TWS18] TORABI F., WARNELL G., STONE P.: Behavioral cloning from observation. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence* (2018), IJCAI'18, AAAI Press, pp. 4950–4957. event-place: Stockholm, Sweden. 12

[vHGS15] VAN HASSELT H., GUEZ A., SILVER D.: Deep reinforcement learning with double q-learning. 8

[WCY*21] WENG J., CHEN H., YAN D., YOU K., DUBURCQ A., ZHANG M., SU H., ZHU J.: Tianshou: a highly modularized deep reinforcement learning library. 21

[WD92] WATKINS C. J. C. H., DAYAN P.: Q-learning. 279–292. 8

[WGH20] WON J., GOPINATH D., HODGINS J.: A scalable approach to control diverse behaviors for physically simulated characters. *ACM Transactions on Graphics (TOG) 39*, 4 (2020), 33–1. 15, 19

[WGSF20] WANG T., GUO Y., SHUGRINA M., FIDLER S.: Unicon: Universal neural controller for physics-based character motion. *arXiv preprint arXiv:2011.15119* (2020). 15, 19

[Wil92] WILLIAMS R. J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. 229–256. 4, 6, 9

[WL19] WON J., LEE J.: Learning body shape variation in physics-based characters. *ACM Transactions on Graphics (TOG) 38*, 6 (2019), 1–12. 15, 19

[WML*17] WU Y., MANSIMOV E., LIAO S., GROSSE R., BA J.: Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. 10

[WOL*21] WERLING K., OMENS D., LEE J., EXARCHOS I., LIU C. K.: Fast and Feature-Complete Differentiable Physics for Articulated Rigid Bodies with Contact. 21

[WRL*21] WANG J., REN Z., LIU T., YU Y., ZHANG C.: QPLEX: Duplex dueling multi-agent q-learning. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021* (2021), OpenReview.net. 13

[WSH*16] WANG Z., SCHAUL T., HESSEL M., HASSELT H., LANCTOT M., FREITAS N.: Dueling network architectures for deep reinforcement learning. In *International Conference on Machine Learning* (2016), PMLR, pp. 1995–2003. ISSN: 1938-7228. 8

[XHLL20] XU D., HUANG X., LI Z., LI X.: Local motion simulation using deep reinforcement learning. 756–779. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/tgis.12620. 17, 19

[XK21] XU P., KARAMOUZAS I.: Human-inspired multi-agent navigation using knowledge distillation. 17, 19

[XLKvdP20] XIE Z., LING H. Y., KIM N. H., VAN DE PANNE M.: ALLSTEPS: Curriculum-driven learning of stepping stone skills. In *Proc. ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (2020). 14, 16, 19

[YHL*20] YANG Y., HAO J., LIAO B., SHAO K., CHEN G., LIU W., TANG H.: Qatten: A general framework for cooperative multiagent reinforcement learning. _eprint: 2002.03939. 13

[YK20] YUAN Y., KITANI K.: Residual force control for agile human behavior imitation and extended motion synthesis. *arXiv preprint arXiv:2006.07364* (2020). 15, 19

[YTL18] YU W., TURK G., LIU C. K.: Learning symmetric and low-energy locomotion. Place: New York, NY, USA Publisher: Association for Computing Machinery. 14, 15, 16, 19

[YVV*21] YU C., VELU A., VINITSKY E., WANG Y., BAYEN A., WU Y.: The surprising effectiveness of PPO in cooperative, multi-agent games. 12

[YWS*21] YUAN Y., WEI S.-E., SIMON T., KITANI K., SARAGIH J. M.: SimPoE: Simulated character control for 3d human pose estimation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021* (2021), Computer Vision Foundation / IEEE, pp. 7159–7169. 15, 19

[YYVDPY21] YIN Z., YANG Z., VAN DE PANNE M., YIN K.: Discovering diverse athletic jumping strategies. Place: New York, NY, USA Publisher: Association for Computing Machinery. 16, 19

[ZLS*20] ZHOU M., LIU Z., SUI P., LI Y., CHUNG Y. Y.: Learning implicit credit assignment for cooperative multi-agent reinforcement learning. 11853–11864. 13

[ZMBD08] ZIEBART B. D., MAAS A., BAGNELL J. A., DEY A. K.: Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3* (2008), AAAI'08, AAAI Press, pp. 1433–1438. event-place: Chicago, Illinois. 12

[ZRS*18] ZAMBALDI V., RAPOSO D., SANTORO A., BAPST V., LI Y., BABUSCHKIN I., TUYLS K., REICHERT D., LILLICRAP T., LOCKHART E., SHANAHAN M., LANGSTON V., PASCANU R., BOTVINICK M., VINYALS O., BATTAGLIA P.: Relational deep reinforcement learning. 17

[ZSSZ18] ZOU H., SU H., SONG S., ZHU J.: Understanding human behaviors in crowds by imitating the decision-making process. 17, 19