Design and Training of Deep Reinforcement Learning Agents

Fabio Pardo

Robot Intelligence Lab
Dyson School of Design Engineering
Imperial College London

Submitted on May 11, 2022, in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Statement of originality

I declare that this thesis was composed by myself, that the work it contains is my own, unless explicitly stated, and that I have acknowledged the contributions of others.

Fabio Pardo
May 11, 2022

Copyright declaration

Acknowledgements

First of all, I would like to thank my thesis supervisor, Petar Kormushev, for giving me the opportunity to freely pursue my PhD and for supporting me throughout this journey. I would also like to express my gratitude to Andrew Davison, my second supervisor, and to Dyson Technology Limited for sponsoring my research. In addition, I would like to express my deepest appreciation to Antoine Cully and Deepak Pathak, for agreeing to be members of my thesis committee. Receiving their insightful opinion on my work is a privilege.

As I began my PhD, I was the only graduate student in the newly formed Robot Intelligence Lab, and research on deep learning and reinforcement learning was rare at Imperial College London. However, over the years, my experience became increasingly rich and joyful as I built a network of friends and peers around me. In particular, I am fortunate to have shared these years with my lab buddies Nemanja Rakićević, Arash Tavakoli, Vitaly Levdik, Digby Chappell, Ke Wang, Roni Saputra, Francesco Cursi, and Ahmad AlAttar, but also with Kai Arulkumaran, Cédric Colas, Manon Flageat, Borja González León, Luca Grillotti, Stephen James, Edward Johns, Rami Kalai, Vittorio La Barbera, Robert Lange, Daniel Lenton, Sriyash Poddar, Pierre Richemond, Aksat Shah, Alex Spies, and Eugene Valassakis.

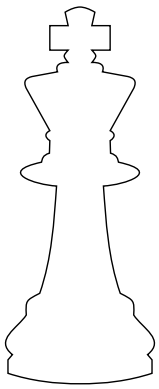
During my two research internships at DeepMind, I have had the wonderful opportunity to meet countless talented researchers and engineers. I am grateful to Raia Hadsell and André Barreto for offering me their supervision, Arthur Guez, Leonard Hasenclever, Nicolas Heess, Peter Humphreys, Timothy Lillicrap, Josh Merel, Mehdi Mirza, and Théophane Weber for their invaluable support, and to Florent Altché, Charles Blundell, Steven Bohez, Lars Buesing, Greg Farquhar, Karol Gregor, Matthew Grimes, Jessica Hamrick, Owen He, Felix Hill, Dan Horgan, Christos Kaplanis, Zita Marinho, Hamza Merzic, Piotr Mirowski, Junhyuk Oh, Laurent Orseau, Dushyant Rao, Andrei Rusu, Tom Schaul, Murray Shanahan, David Silver, Yuval Tassa, Denis Teplyashin, Hado van Hasselt, Petar Veličković, Fabio Viola, Greg Wayne, Daan Wierstra, and Martina Zambelli for their thoughtful attention.

I am also grateful to my friends outside of work, who allowed me to take a break and enjoy my free time, in London and Paris, and in particular, to Nafie Bakkali, Christine Barrere, Mathieu Barrere, William Bastien, Timoté Bertrand, Benjamin Beyret, Julien Bouvier, Corentin Bréhard, Nicolas Bruneel, Coline Ciresa, Gaultier Ciresa, Christelle Cottier, Claire Couturier, Roland Ding, Julie Duwat, Alexandre Gaillet, Arnaud Goffin, Edouard Homberg, Philippe Inthavong, Gabriel Kheradmand, Annie Li, Alain Lite, Alice Lor, Victor Ly, Vincent Ly, Mélanie Martin, Paul Müller, Brandon Neang, Gabrielle Nguyen, Kyriacos Nikiforou, Mai Phan, David Salmon, Émilie Sanchez, Thomas Sanchez, Leslie Tang, Véronique Tran, and Christophe Vandrot.

I will be forever grateful to my family, and in particular to my parents Anna Elli-Pardo and Fabrice Pardo, my French grandparents Myrielle and Bruno Pardo, my Italian grandparents Giuseppina and Virginio Elli, my uncle Pierre-Emmanuel, aunt Claire, and cousins Alice and Noé Pardo. They are the foundations that allowed me to grow. They taught me to care for others, be ambitious, seek knowledge, think, and remain strong in all circumstances. I am extremely grateful to my sister Sara Pardo-Rey and my brother-in-law Pierre-Antoine Rey for their support and for giving birth to my beloved niece Alessia and nephew Elio Rey. They are, without a doubt, the most adorable children on the planet. I would also like to express my gratitude to my extended family, Mireille and Yvon Rey, as well as Huong, Charles and Alexandre Nguyen and their relatives in Vietnam. Their warm and exuberant welcome on all occasions has never failed to touch me. I would also like to thank Elsa Godart and Rosa Gomes, for their presence and kindness towards our family. I am also grateful to all the animals I have had in my life, Diane, Sissi, Filo, Neo, and Leiko. Their love, happiness, and appreciation of simple things are true lessons for us humans. In addition, I cannot help but mention how hard it was to lose some of my loved ones over the past few years. I believe they would have been proud of me today. Maman, Dadou, Mamée, Nonno, Nonna, et Mireille, vous me manquez. Mi mancate.

Finally, one person deserves my most sincere gratitude, my partner in life, Cécile Nguyen. Her love and support have been invaluable over the past 13 years. She has endured, more than anyone, the side effects of my aspirations. The completion of my PhD marks the end of an important chapter in both of our lives. Wherever we go next, life will be beautiful because we will be together. Je t'aime ma chérie.

On May 11, 1997, exactly 25 years ago, IBM's Deep Blue defeated Garry Kasparov in a six-game chess match. It was the first time a machine was able to surpass a reigning world champion under tournament conditions. Artificial intelligence was making history.



Abstract

Deep reinforcement learning is a field of research at the intersection of reinforcement learning and deep learning. On one side, the problem that researchers address is the one of reinforcement learning: to act efficiently. A large number of algorithms were developed decades ago in this field to update value functions and policies, explore, and plan. On the other side, deep learning methods provide powerful function approximators to address the problem of representing functions such as policies, value functions, and models. The combination of ideas from these two fields offers exciting new perspectives. However, building successful deep reinforcement learning experiments is particularly difficult due to the large number of elements that must be combined and adjusted appropriately. This thesis proposes a broad overview of the organization of these elements around three main axes: agent design, environment design, and infrastructure design. Arguably, the success of deep reinforcement learning research is due to the tremendous amount of effort that went into each of them, both from a scientific and engineering perspective, and their diffusion via open source repositories. For each of these three axes, a dedicated part of the thesis describes a number of related works that were carried out during the doctoral research.

The first part, devoted to the design of agents, presents two works. The first one addresses the problem of applying discrete action methods to large multidimensional action spaces. A general method called action branching is proposed, and its effectiveness is demonstrated with a novel agent, named BDQ, applied to discretized continuous action spaces. The second work deals with the problem of maximizing the utility of a single transition when learning to achieve a large number of goals. In particular, it focuses on learning to reach spatial locations in games and proposes a new method called Q-map to do so efficiently. An exploration mechanism based on this method is then used to demonstrate the effectiveness of goal-directed exploration. Elements of these works cover some of the main building blocks of agents: update methods, neural architectures, exploration strategies, replays, and hierarchy.

The second part, devoted to the design of environments, also presents two works. The first one shows how various tasks and demonstrations can be combined to learn complex skill spaces that can then be reused to solve even more challenging tasks. The proposed method, called CoMic, extends previous work on motor primitives by using a single multi-clip motion capture tracking task in conjunction with complementary tasks targeting out-of-distribution movements. The second work addresses a particular type of control method vastly neglected in traditional environments but essential for animals: muscle control. An open source codebase called OstrichRL is proposed, containing a musculoskeletal model of an ostrich, an ensemble of tasks, and motion capture data. The results obtained by training a state-of-the-art agent on the proposed tasks show that controlling such a complex system is very difficult and illustrate the importance of using motion capture data. Elements of these works demonstrate the meticulous work that must go into designing environment parts such as: models, observations, rewards, terminations, resets, steps, and demonstrations.

The third part, on the design of infrastructures, presents three works. The first one explains the difference between the types of time limits commonly used in reinforcement learning and why they are often treated inappropriately. In one case, tasks are time-limited by nature and a notion of time should be available to agents to maintain the Markov property of the underlying decision process. In the other case, tasks are not time-limited by nature, but time limits are used for convenience to diversify experiences. This is the most common case. It requires a distinction between time limits and environmental terminations, and bootstrapping should be performed at the end of partial episodes. The second work proposes to unify the most popular deep learning frameworks using a single library called Ivy, and provides new differentiable and framework-agnostic libraries built with it. Four such code bases are provided for gradient-based robot motion planning, mechanics, 3D vision, and differentiable continuous control environments. Finally, the third paper proposes a novel deep reinforcement learning library, called Tonic, built with simplicity and modularity in mind, to accelerate prototyping and evaluation. In particular, it contains implementations of several continuous control agents and a large-scale benchmark. Elements of these works illustrate the different components to consider when building the infrastructure for an experiment: deep learning framework, schedules, and distributed training. Added to these are the various ways to perform evaluations and analyze results for meaningful, interpretable, and reproducible deep reinforcement learning research.

Table of contents

List of figures

List of tables

List of code snippets

Chapter 1

Introduction

1.1 Intelligent machines

Life-like objects For millennia, mankind has been fascinated by the idea that statues or machines could be brought to life. Mythology and folklore are full of such tales, including living metal statues built by Hephaestus, the god of blacksmiths and fire, the golem, a creature made of mud, and medieval bronze heads that could answer any question put to them. Instances of life-like machinery escaped fiction to become real attractions during the 18th century with the construction of sophisticated *automata*. Notable examples include the digestive duck (1739) and the chess-playing mechanical Turk (1770). The latter could play chess up to master level with human-like gestures, fooling many scholars and kings of the time, until it was revealed that a person was hidden inside. As the belief in magic and alchemy was replaced by rationality and the scientific method, people realized that extraordinary craftsmanship was not enough and technology was lacking.

Programmable computers The first concept of a programmable computer appeared with Charles Babbage's *analytical engine*. Although this complex machine was never built, it was in principle based on a memory and a computing unit, with programs provided on punched cards. With the new perspectives opened by this invention, Babbage and Ada Lovelace fantasized about the capabilities of such a machine. Lovelace wrote in a memoir (1842), that this machine "might act upon other things besides number". They both knew that mankind was now flirting with mechanized intelligence. Nearly a decade later, Alan Turing proposed a mathematical model of computation that relies on a strip of symbols, a moving read/write head, and a table of rules (Turing, 1936). This model, known as the *Turing machine*, was a key step in the development of ENIAC (electronic numerical integrator and computer), the first true programmable, electronic, general-purpose digital computer. This "giant brain", as the press called it at the time, was indeed very large and mainly

built for military simulations. As the development of programmable computers became a reality, Turing pondered the prospect of intelligent machines and wondered how one could decide whether a machine was truly intelligent. In an article entitled "Computing machinery and intelligence" (Turing, 1950), he proposed a test originally called *imitation game*, better known as the *Turing test*, in which a human evaluator would converse with a real person and a computer. If the evaluator could not reliably distinguish the machine from the human, the machine would pass the test, thus proving its intelligence.

Artificial intelligence In the early 1950s, various researchers worked on fields related to "thinking machines" under different names: cybernetics, automata theory, and complex information processing. John McCarthy, proposed to organize a group to clarify and develop ideas, and picked the name *artificial intelligence* (AI). The famous Dartmouth workshop of 1956 is widely recognized as the birth place of this new field, defined as the study and development of computer systems capable of performing tasks typically associated with intelligent beings. However, the definition of intelligence remains rather vague. The term refers to the knowledge and intellectual skills required to produce non-trivial behavior. A program designed to play Tic-Tac-Toe perfectly may be considered more intelligent than a less optimal one, while being completely unable to play another game or solve another task. Early efforts were largely devoted to symbolic computation. Reasoning was considered the pinnacle of intelligence, and the prospect of creating a program that could outperform humans at chess was considered by many to be the Holy Grail of AI. In 1997, Deep Blue, a chess computer developed by IBM, was the first machine to defeat the reigning world champion, Garry Kasparov, under tournament conditions. However, this obviously did not mean that intelligence was solved. A flagrant example comes from robotics, where symbolic manipulation could be used to develop plans such as a sequence of picking up and dropping off objects, but interestingly, researchers noticed that producing dexterous movements with robots was much harder. Hans Moravec wrote "it is comparatively easy to make computers exhibit adult-level performance in solving problems on intelligence tests or playing checkers, and difficult or impossible to give them the skills of a one-year-old when it comes to perception and mobility" (Moravec, 1988).

Machine learning Hard-coded programs lack generality, and tasks requiring a large number of interacting factors are simply too complex for traditional coding approaches. The solution came with the realization that, in many cases, it is easier to create a more general program that can improve its performance over time, than to

solve the task at hand directly. Naturally, this automatic adaptation was compared to biological learning and the term *machine learning* was chosen to describe such algorithms. The simplest type of machine learning is called *supervised learning*. Algorithms in this category use a dataset of input and output pairs and are trained to map the former with the latter. This mapping function, called a model, is the trainable part, and a loss function specifies how close the predicted output is to the correct one. Most supervised learning approaches use a differentiable parametric model and a differentiable loss. By taking the gradient of the loss with respect to the model parameters, and updating the parameters in small steps in the reverse direction of the gradient, the predictions improve. This is called gradient descent. The second category of machine learning, called *unsupervised learning*, is used when target outputs are not available but some structure in the data needs to be discovered. Algorithms in this category are trained to represent the data. They encompass a variety of approaches such as clustering, prediction of missing labels, or finding low-dimensional representations, and can also rely on differentiable losses. Both categories of machine learning rely on fairly simple experimental setups with well-defined datasets. However, the type of learning most apparent in nature is arguably different. Animals learn by interacting with their environment. They learn to make predictions and take actions to achieve certain goals, such as escaping danger, eating, resting, reproducing, or just having fun. In these cases, supervision is not available and there is no clear loss function to directly assess the myriad of actions taken and their influence on the environment. A third type of machine learning exists specifically to emulate this kind of learning: *reinforcement learning*.

Reinforcement learning To the problem of sequential decision making, reinforcement learning (RL) proposes a particularly clever strategy: since target actions are not available, a *reward signal* is used instead to drive the objective. This signal does not directly describe the quality of an action produced in a given situation, but may be the result of decisions taken long ago and credit must be assigned accordingly. The objective is not to maximize the next immediate reward, but rather the accumulation of these in the future, called a return. Even if actions lead to modest immediate rewards, they can be valuable if they lead to greater ones in the more distant future. The RL framework is generally presented as that of an agent that must learn to interact with its environment, with inputs being observations and outputs being actions sent to the environment. The environment is modeled as a *Markov decision process* with transitions from one state to another, depending on the previous state and action. The environment is similar to a dataset that the agent has to sample through exploration, in order to discover useful information.

The collected data is distilled into functions that typically represent expected values, policies, and models of the environment. While early research in the field used simple tables to hold mapping values, parametric functions have made it possible to represent increasingly more powerful features.

Deep learning The first practical example of parametric machine learning is the *perceptron*, invented by Frank Rosenblatt in 1958. The Mark I perceptron was built to be an image recognition machine. It had an array of 400 photocells, randomly connected to *neurons*, themselves linearly combined to produce a binary classifier. The perceptron initially attracted a lot of interest, with the New York Times reporting it to be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence". However, a single perceptron was only capable of learning linearly separable models and the controversy grew, mainly fueled by the book "Perceptrons" by Marvin Minsky and Seymour Papert (Minsky & Papert, 1969). This disillusion and a number of other cycles of hype and disappointment led to repeated significant cuts in fundings, called "AI winters". Research on artificial neural networks has nevertheless continued to make progress, and in particular, it was shown that a stack of at least two perceptrons with a non-linearity in between was powerful enough to represent arbitrarily complex functions, and that such deeper models could be trained efficiently by gradient descent with a simple chain rule, a method called *backpropagation*. The work of Yann LeCun on convolutional neural networks, a type of network loosely inspired by the visual cortex, trained end to end by backpropagation to recognize hand-written numbers (LeCun et al., 1989) was applied by several banks to read checks automatically. However, despite this success, interest in the field of artificial neural networks remained low until a major breakthrough. In 2012, a model based on a deep stack of convolutions called AlexNet (Krizhevsky et al., 2012), won the ImageNet Large Scale Visual Recognition Challenge with more than 10.8 percentage points lower error than that of the runner up, relying on more traditional approaches with handcrafted features. This milestone showed that a combination of large datasets, a deep neural networks, end-to-end training via backpropagation of gradient, and specialized hardware (GPU) could be a game changer. Interest in neural networks grew again quickly and the field of deep learning was reborn. An important part of the success of the deep learning field has been the recurrent release of open-source code bases for building and training models such as Theano (2011), Torch7 (2013), TensorFlow (2015), PyTorch (2016), and JAX (2018), but also datasets such as MNIST (1998), CIFAR (2009), ImageNet (2009), Sentiment140 (2009), COCO (2015), and Amazon reviews (2015).

Deep reinforcement learning Reinforcement learning can greatly benefit from powerful parametric function approximators. A good early example of an agent combining reinforcement learning techniques and a deep neural network was TD-Gammon (Tesauro, 1994). This program learned to play the game of Backgammon solely by playing against itself from a combination of board game description and hand crafted features, to reach strong master level. However, one can argue that the field of *deep reinforcement learning* was truly born with the paper "Playing Atari with deep reinforcement learning" (Mnih et al., 2013). This article presented *Deep Q-Network* (DQN), the first algorithm able to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. This paper and its improved version (Mnih et al., 2015), introduced a number of methods for designing agents, environments and infrastructures that remained standard to this day. The performance of DQN was at human-level or above on a number of Atari 2600 games from pixels. Furthermore, this work was accompanied by the source code and gave a starting point for the deep reinforcement learning community to grow. Other equally impressive milestones followed, including AlphaGo (Silver et al., 2016), the first computer program to defeat one of the very best Go players, Lee Sedol, and AlphaStar (Vinyals et al., 2019), the first AI to reach grandmaster level in the StarCraft II game. Deep reinforcement learning research heavily relies on deep RL libraries such as rllab (2016), OpenAI Baselines (2017), Dopamine (2018), TensorFlow Agents (2018), Spinning Up (2018), and Acme (2020), but also environment suites such as the Arcade Learning Environment (2012), OpenAI Gym (2016), PyBullet (2016), and dm_control (2017).

1.2 My personal journey into AI

1991 - 2009: Preamble I was born in 1991 in Vitry-sur-Seine, a city in the southern suburbs of Paris, France. As a child, I was fascinated by robots and spent hours sketching them. As I grew up, I tried to develop my interest in drawing and painting but my curiosity in the arts evolved into a passion for music. I learned to play piano, bass and guitar with different groups of friends and as I began to compose my own music, I decided to get more and more involved in this creative process. After graduating high school in 2009, I wanted to make music professionally.

2009 - 2012: Bachelor of science in computer science In parallel to my efforts to compose music, I studied computer science at the Pierre and Marie Curie University (Paris VI). I learned to program for the very first time and immediately fell in love with this new way of expression. I wanted to learn as many languages

as possible such as Scheme, C, C++, Java, JavaScript, OCaml, Python, Visual Basic, and Assembly. But more important than languages, I saw in programming an opportunity to create intelligent programs. When I started studying AI, I participated and was twice a finalist in the French AI competition Prologin, which consisted of algorithmic testing and a 36-hour hackathon on AI for multiplayer games. However, while I loved some of the classic tools of AI, like logic and planning, it became very clear to me that machine learning was the key to creating more intelligent programs. In particular, I was fascinated by the concept of strong AI, an artificial intelligence with intellectual capabilities functionally equal to or greater than those of a human.

2012 - 2014: Master of science in cognitive science After obtaining my Bachelor's degree, I decided to study cognitive science to have a better grasp at our understanding of brains and minds. I enrolled in a course of study called Cogmaster, between three schools: École Normale Supérieure (ENS) Ulm, École des Hautes Études en Sciences Sociales (EHESS), and Paris Descartes University (Paris V). There, I studied computational neuroscience, experimental psychology, neuroimaging, neuroanatomy, and had my first contact with reinforcement learning and the perceptron algorithm. I did a research internship with Jean-Gabriel Ganascia on ontology visualization methods and their impact on human memorization. The hypothesis was that presenting structured data as maps of knowledge islands with visual support for locating information would facilitate recall. Importantly, during one of our discussions about strong intelligence, I discovered that the contemporary term for strong AI was *artificial general intelligence* (AGI) and that there was a community of researchers working specifically on this topic. I then contacted Laurent Orseau, one of the only French researchers publishing at the AGI conference, who offered me a research internship. I worked on optimal decision making based on a mixture of prediction experts and on the idea of a homeostatic engine for reinforcement learning agents. Importantly, Laurent showed me the work of Marcus Hutter, Jürgen Schmidhuber, and Shane Legg on formalizing general intelligence and proposing algorithms to achieve it. At the end of my Masters, I did a research internship with David Filliat in the FLOWERS team, working on developmental robotics. In particular, I fully developed an experimental setup to study the emergence of multimodal concepts for a humanoid robot in interaction with a human tutor. The setup was particularly ambitious and included the perception of objects and gestures of the tutor, control of the movements of the head of the robot, speech to text, and text to speech, representation of knowledge with a non-negative matrix factorization, visualization of learned concepts on screen, recruitment and experimentation with volunteers, and analysis of the results.

2014 - 2015: Master of science in computer science After receiving my Master's degree in cognitive science, I wanted to strengthen my knowledge in AI and robotics and enrolled in a second Master's degree, at the Pierre et Marie Curie University, called ANDROIDE. In the meantime, I was contacted by a recruiter from Google who proposed me to join the Machine Intelligence team and I started to prepare for technical interviews in algorithms and data structures. Unfortunately, while the recruiting process was successful, I was not granted a H-1B visa to go work in the United States. At the same time, I was selected for a research internship in Tokyo at the National Institute of Informatics (NII) with Tetsunari Inamura, where I started working on SIGVerse, a cloud-based VR platform for sharing social and embodied skills between humans and robots. After implementing an efficient motion planning algorithm in SIGVerse, I decided to work on a project more focused on learning. At the time, DeepMind had just published the DQN article in Nature (Mnih et al., 2015) and released its source code in Lua and Torch. I was fascinated by this new approach based on deep reinforcement learning, which managed to achieve human-level performance on a wide range of Atari 2600 games. I therefore spent the rest of my internship adapting the DQN code to a navigation task in SIGVerse with moderate success, but importantly, this was my first experience with deep reinforcement learning.

2015 - 2016: Interlude After graduating with my second Master's degree, I applied for a research engineer position at DeepMind. After several successful technical interviews, I had the chance to speak with Shane Legg, one of the founders of the company, whose thesis "Machine super intelligence" I had read a few years earlier. Since the proposed role was on the safety side of AGI and did not match my interests, my application was not processed further. Having decided to pursue a PhD, I applied for a proposal submitted by Petar Kormushev at Imperial College London and was lucky enough to be accepted.

2016 - 2017: Early stage of the PhD I moved to London and started my PhD research. The lab I was joining, the Robot Intelligence Lab, had just been created and I was the first PhD student. My first task was to help set up our DE NIRO platform, a custom robot consisting of a Baxter torso attached to an electric wheelchair, with camera sensors on the head and a laptop computer on the back. In particular, I worked with Rami Kalai on hacking the control system of the electric wheelchair to replace the original joystick with a custom wireless one, allowing movements to be sent by an Xbox controller or by software. A lot of time was also invested in public demonstrations of our robotic platform, at the university,

in museums and other exhibitions. I also helped start the deep RL reading group at Imperial, which I then organized for four years. I started working as a graduate teaching assistant in computer science, teaching Python and later robotics. During my early stage assessment, I proposed to focus my research on goal-directed RL. In particular, I was interested in developing a hierarchical approach with a task agnostic low-level controller and and task-specific high-level planner. For training the task-agnostic low-level controller, I proposed to use states encountered during training as counterfactual goals during replay, and guide exploration by selecting less visited goals. However, the publication of a number of similar ideas, including hindsight experience replay (Andrychowicz et al., 2017) made me reconsider my project.

2017 - 2018: First publications As I was rethinking some of my ideas, I joined a project led by Arash Tavakoli on adapting DQN to multidimensional discrete action spaces. Arash had an initial code-base and promising results. I helped formalizing the action branching idea, improving the proposed agent, reformatting the code-base, scheduling experiments, and writing a paper that was later presented at AAAI 2018 (Tavakoli et al., 2018). Going back to my initial ideas on goal-reaching policies, I started to wonder about the notion of time limits. In particular I thought that while interacting with its environment, an agent should be given some time to reach a given goal but the collected transitions should be useful to learn to reach other types of goals off-policy and without time constraints. Noticing that there were no existing works on how to handle time properly in reinforcement learning, I started to identify different scenarios and had a few promising experiments running. I then proposed to Arash and Vitaly Levdik to join the project. Together, we better formalized the concepts, ran experiments and wrote a paper that was later presented at ICML 2018 (Pardo et al., 2018). I then continued to explore research ideas such as adapting action branching to the Rainbow agent (Hessel et al., 2018) from Dopamine (Castro et al., 2018) for Atari 2600 games, applying hierarchical RL to Montezuma's Revenge, benchmarking deep RL agents with the embryo of a self-made deep RL library, and leveraging recurrent neural networks to represent behaviour patterns, in particular from humanoid motion capture data. I also went back to the idea of updating a goal-oriented policy towards all goals simultaneously for every transition. Early results were promising on Montezuma's Revenge and as Vitaly joined the project, we thought of using the proposed algorithm called Q-map for a novel exploration strategy. A paper based on this work was later presented at AAAI 2020 (Pardo et al., 2020). During this period, I also decided to look for internship opportunities. After a few conversations with Raia Hadsell and a long application process with

technical interviews, I had the chance to be accepted for a research internship in DeepMind's deep learning team.

2019: Internship at DeepMind Joining DeepMind for a research internship was a fantastic experience. I took every opportunity to meet researchers and engineers, discuss ideas, and learn about the tools developed in-house. Overall, I had a great internship under Raia's supervision. I mostly worked on hierarchical RL, and in particular, I initially tried to pursue some of the ideas I had about storing behavior codes in recurrent neural networks. I was working mainly with Raia, Nicolas Heess, Josh Merel and Leonard Hasenclever. As the results obtained with the behavior codes did not match my expectations, I decided to try other ideas. I proposed a second project involving the design of competitive self-play environments to force the emergence of useful parkour skills. As I became more familiar with the neural probabilistic motor primitives (Merel et al., 2018b) used in the team, I then proposed a third project consisting in adapting the required complex and rigid training pipeline to a full RL one with an improved tracking setup. Just as the results were starting to come in, the internship ended and I could not continue to run experiments. Fortunately, the team, especially Leonard, continued to work on this last project and expanded it with additional tasks. I continued to contribute as much as I could by proposing ideas and helped write a paper that was later presented at the ICML 2020 conference (Hasenclever et al., 2020).

2019 - 2021: Various projects and a pandemic After completing the internship, I worked on expanding the deep RL reading group at Imperial and proposed to help with controlling a robot developed in the lab called Slider. I built a fast simulation of the robot and showed that it could be controlled with RL instead of more traditional motion planning methods. I then worked for a few weeks on the idea to represent entire RL algorithms such as Q-learning in recurrent networks, without using gradient descent or replay buffers. While some early results were promising I found that the computation power required would be a too heavy bottleneck. I also spent some time trying to solve an issue found with value-gradient methods in continuous control, when a policy is trapped in a local optimum even though the critic is aware of higher-value actions. My idea relied on sampling the critic and increasing the likelihood of the best actions but this exact same idea was concurrently published under the name amortized Q-learning (Van de Wiele et al., 2020). Inspired by model-free planning (Guez et al., 2019), I spent some time investigating how the computation of an output can be amortized over multiple thinking steps in supervised learning tasks such as calculus, rephrasing supervised learning tasks as reinforcement learning ones.

I was also inspired by the recent success of Transformer architectures in modeling language, and thought about ways to adapt their attention mechanisms to develop plans. I also increased my understanding of how multilayer perceptrons represent mappings by creating neurons that fire at specific locations, similar to grid cells, and I tried to use this to develop special neurons similar to radial basis functions. Around that time, the SARS-CoV-2 virus started to spread around the world, becoming a pandemic. The university closed, I moved my workstation to my apartment and started working on more engineering-oriented projects as collaboration with other students was limited. I started by working on a deep RL library from scratch, to allow me to experiment with my various ideas. After four months of work, I released this project under the name Tonic RL Library, with an accompanying paper (Pardo, 2020). While still working from home, Daniel Lenton proposed me to join a project on a new library to unify existing deep learning frameworks. He had some proof of concept relying on type checking. I helped him rethink the core functions and framework specification and I developed differentiable environments for gradient-based planning. We released this new library under the name Ivy shortly afterwards, with an accompanying paper (Lenton et al., 2021). I was also contacted by Vittorio La Barbera, who was working on a musculoskeletal model of an ostrich and wanted to control it using deep RL. We started a colaboration that took a year of hard work, both improving greatly the model and adding reinforcement learning tasks and electromyography (EMG) comparisons. Ultimately, the project was released under the name OstrichRL (La Barbera et al., 2021). I was also contacted by Sriyash Poddar, an undergraduate student at the Indian Institute of Technology Kharagpur, who could not find an internship due to the pandemic. We worked together for several months on various projects. In particular, I was inspired by the idea of a single policy able to control various morphologies (Huang et al., 2020b, e.g.). We developed environments allowing us to randomize bodies and a Transformer-based architecture that would take a morphology description and sensor readings in input and would output torque values. During this strange period dictated by the cycle of lockdowns and restrictions due to the pandemic, I went back to France and looked for a new research internship opportunity. I was first offered an opportunity at Sony AI in Tokyo. Unfortunately the start date was postponed several times because of the Japanese borders remaining closed and I never had a chance to actually start it. I also looked for a second opportunity at DeepMind and managed to get one in the RL team with André Barreto that would be entirely remote, from London.

2021: Second internship at DeepMind During this second internship, I explored various research ideas around the theme of learning to think. I mostly

collaborated with André, Théophane Weber, Arthur Guez, Peter Humphreys, Timothy Lillicrap, and Mehdi Mirza. I first continued the work I already started on learning to solve supervised questions with adaptive thinking time but then switched to planning using search trees. More specifically, I created a reinforcement learning task involving a search policy that selects which expansions to perform using attention. At the time of writing this thesis, this project is still in progress.

2022: End of the PhD Looking backward, I realize that my various research projects have allowed me to collect a vast amount of knowledge in deep learning, reinforcement learning, and deep reinforcement learning. But more importantly, these years of doctoral studies have given me the skills to conduct my own research. I now wish to find a position that will allow me to fulfill my curiosity for intelligent machines, and I look forward to seeing what artificial intelligence research will uncover in the future.

1.3 Publications

The following is a list, in chronological order, of the successful research I completed during my doctoral studies. My contributions are stated for the works where I am not the first author.

1. **Action branching architectures for deep reinforcement learning**.
   Arash Tavakoli, **Fabio Pardo**, and Petar Kormushev.
   AAAI Conference on Artificial Intelligence (Tavakoli et al., 2018).
   This work is discussed in Chapter 4.
   I joined the project when preliminary results already existed, but then re-implemented a large part of the code base, contributed to the main ideas, experimentations, and writing of the manuscript.

2. **Time limits in reinforcement learning**.
   Fabio Pardo, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev.
   International Conference on Machine Learning (ICML) (Pardo et al., 2018).
   This work is discussed in Chapter 10.

3. **Scaling all-goals updates in reinforcement learning using convolutional neural networks**.
   Fabio Pardo, Vitaly Levdik, and Petar Kormushev.
   AAAI Conference on Artificial Intelligence (Pardo et al., 2020).
   This work is discussed in Chapter 5.

4. **CoMic: Complementary task learning & mimicry for reusable skills**.
   Leonard Hasenclever, **Fabio Pardo**, Raia Hadsell, Nicolas Heess, and Josh Merel.
   International Conference on Machine Learning (ICML) (Hasenclever et al., 2020).
   This work is discussed in Chapter 7.
   I started this project during an internship at DeepMind. Neural probabilistic motor primitives (NPMP) already existed but required a cumbersome pipeline and did not allow for online discovery of new skills. I showed that multi-clip tracking was possible using a single reinforcement learning task I developed and suggested combining tracking with other tasks. After the internship, I participated in meetings to discuss the results and organize the research, and contributed to the writing of the manuscript.

5. **Tonic: A deep reinforcement learning library for fast prototyping and benchmarking**.
   Fabio Pardo.
   NeurIPS deep RL workshop (Pardo, 2020).
   This work is discussed in Chapter 12.

6. **Ivy: Templated deep learning for inter-framework portability**.
   Daniel Lenton, **Fabio Pardo**, Fabian Falck, Stephen James, and Ronald Clark.
   arXiv preprint 2102.02886 (Lenton et al., 2021).
   This work is discussed in Chapter 11.
   I joined this project when a proof of concept for a single API covering multiple frameworks already existed, but proposed to reformat most of the the core functions of the library, implemented differentiable environments, and contributed to the writing of the manuscript.

7. **OstrichRL: A musculoskeletal ostrich simulation to study bio-mechanical locomotion**.
   Vittorio La Barbera*, **Fabio Pardo***, Yuval Tassa, Monica Daley, Christopher Richards, Petar Kormushev, and John Hutchinson (* equal contribution).
   NeurIPS deep RL workshop (La Barbera et al., 2021).
   This work is discussed in Chapter 8.
   I joined this project when a preliminary planar model existed, but then implemented a proof of concept motion capture tracking task, cleaned the motion capture data with a recurrent network, contributed in equal parts to extending the model to a full 3D one with a realistic neck, implementing the reinforcement learning tasks, running experiments, and writing of the manuscript.

1.4 Thesis overview

The various projects I worked on during my PhD touched on a wide range of topics in the field of deep reinforcement learning. These different works allowed me to gain a profound understanding of how to construct experiments and produce research in the field. In this thesis, instead of focusing on a single research question, I chose to convey my broader knowledge of how research is conducted. Specifically, I explain that the many parts that constitute experiments can be naturally grouped around three main axes: agents, environments, and training infrastructures. My research is therefore not presented in a chronological order but rather grouped to illustrate these axes one by one, while mixing elements of theoretical and experimental research with engineering.

Part I focuses on the design of agents. It begins with Chapter 3, which describes what we mean by an agent in reinforcement learning and how modern deep reinforcement learning agents can be described in terms of multiple constituting components. Two works on agent design follow. Chapter 4 is based on the paper "Action branching architectures for deep reinforcement learning" (Tavakoli et al., 2018), and Chapter 5 is based on the paper "Scaling all-goals updates in reinforcement learning using convolutional neural networks" (Pardo et al., 2020).

Part II is devoted to the environment design aspect. It starts with Chapter 6, which explains what we mean by an environment in reinforcement learning and how modern deep reinforcement learning environments can also be described in terms of multiple constituents. Following the same structure as for the previous part, two works on environments are presented. Chapter 7 is based on the paper "CoMic: Complementary task learning & mimicry for reusable skills" (Hasenclever et al., 2020), and Chapter 8 is based on the paper "OstrichRL: A musculoskeletal ostrich simulation to study bio-mechanical locomotion" (La Barbera et al., 2021).

Part III centers on the design of training infrastructures. It begins with Chapter 9, describing how agents and environments interact in modern deep reinforcement learning libraries, and explaining what are the constitutive components of the infrastructures. For this last part, three works are presented. Chapter 10 is based on the paper "Time limits in reinforcement learning" (Pardo et al., 2018), Chapter 11 is based on the paper "Ivy: Templated deep learning for inter-framework portability" (Lenton et al., 2021), and Chapter 12 concludes this sequence of works and is based on the paper "Tonic: A deep reinforcement learning library for fast prototyping and benchmarking" (Pardo, 2020).

Chapter 2

Background

In this chapter, some prerequisite knowledge is provided in reinforcement learning, deep learning, and deep reinforcement learning. This provided background is far from exhaustive and is specific to the research proposed in this thesis. For more general overviews, books such as Sutton & Barto (2018) and Szepesvári (2010) for reinforcement learning, and Goodfellow et al. (2016) for deep learning are great starting points.

2.1 Reinforcement learning

2.1.1 Framework

Markov decision process In the standard reinforcement learning framework, an agent interacts sequentially with its environment. The mathematical framework used to model the decision making process is called Markov decision process (MDP) and is intended to describe the three aspects required for decision making: sensation, action, and goal. MDPs are an extension of Markov chains, adding actions for choice and rewards for goals. An MDP is usually described by a state space $\mathcal{S}$, an action space $\mathcal{A}$, a reward function $r(s, a, s')$, and a state-transition distribution $p(s'|s, a)$. At each time step t, the agent is given a state s_t and generates an action a_t. The environment responds by providing a new state s_{t+1} and a reward r_{t+1}. Some states can be terminal, meaning that no further interaction is possible after reaching them. A decision maker uses a policy, denoted π, to map states to probabilities over actions. In MDPs, states are called Markovian because they contain all the information required to model state transitions and rewards, and therefore to represent optimal policies. No other information, such as an history of previous states, is required. The internal states used by an environment do not need to be the same as those provided to an agent. For example, the full state of a physics simulator might include parameters, such as friction coefficients or the value of the gravity acceleration, that

do not need to be communicated to the agent if they remain unchanged. Similarly, the state of objects, including their coordinates, rotations, and velocities, can be described in various ways and from different reference points while conveying the same information.

Partially observable Markov decision process Inputs to an agent are often called observations. While observations are usually described as Markov states in reinforcement learning algorithms, they are often non-Markovian in practice. The resulting decision process is called a partially observable Markov decision process (POMDP) (Lovejoy, 1991). Various methods have been proposed in the literature to either explicitly estimate the probability distribution of the underlying Markov states or to aggregate past observations with recurrent neural networks to implicitly reconstruct Markov states. In the remainder of this chapter, we assume that observations are Markov states for simplicity.

Episodes The interaction between the agent and the environment usually takes place in a finite number of steps, constituting episodes. First, the initial state s_0 is sampled from a distribution $p_0(s_0)$, then the agent perceives this state and outputs a first action a_0. Then, the environment transitions to the next state s_1, based on s_0 and a_0 and generates a reward r_1. The agent perceives this new state and reward, and outputs a new action a_2. The interaction loop continues like this until the episode terminates by reaching a terminal state. For a path-finding task, a terminal state can be the end of a maze, for a video game, it can be a fatal blow from an enemy, and for a robot, it can be losing balance and falling.

2.1.2 Objective

Sum of rewards The objective of the decision maker is to construct a policy π that maximizes some cumulative function of the future rewards, a *return* R. Directly using the sum of rewards can present multiple issues however. For example, if episodes can last indefinitely, an infinite number of rewards can be collected making the sum unbounded. In other situations, even if the number of rewards is limited, there is no incentive to collect them sooner than later because the sum is agnostic to time.

Discounted sum of rewards Instead of using a simple sum of rewards, the most common transformation applied to them is a geometric progression using a discount factor $\gamma \in [0, 1)$. This transformation solves the two issues listed above: it makes the sum of discounted rewards bounded, and it makes short-term rewards more

appealing than long-term ones. However, it is worth noting that a small discount factor can make policies too myopic, eagerly preferring small immediate rewards to larger delayed ones. In the rest of this thesis, unless stated otherwise, we will assume that returns are discounted.

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

2.1.3 Policy evaluation

State value function The expected return that a policy π will generate from a given state s is called a state value function and is defined as:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(.|s),s'\sim p(.|s,a)}\left[r(s,a,s') + \gamma V^\pi(s')\right]$$

This equation is called a *Bellman equation* as it connects the value of a state to the values of the successor states.

To improve a parametric approximation V_θ towards the true state value function V^π of a policy π, a simple mean squared error (MSE) is often taken between predicted values and estimates. Given a state s and a return R, the loss is defined as:

$$L_\theta(s,R) = \frac{1}{2}\left(V_\theta(s) - R\right)^2$$

and by taking its gradient with respect to the parameters θ, we get:

$$\nabla_\theta L_\theta(s,R) = \left(V_\theta(s) - R\right)\nabla_\theta V_\theta(s)$$

The return R is an approximation of the expected return from s when following π. If the transition and reward functions are known, or approximated with a model, the one-step expectations in the Bellman equation can be computed directly. This method is known as *dynamic programming* (DP). Most often, however, R is a sample-based estimate. It can be the discounted sum of rewards perceived by the agent in the remainder of the episode. This is called a *Monte Carlo return* and is subject to a large variance because it estimates the expectation with a single sample. To reduce this variance and exploit the recursive nature of Bellman equations, R can be constructed as a mixture of sampled values and predictions. For example, an *n-step return* is defined as a Monte Carlo return up to some length, and a prediction, called bootstrap, for the remainder, like so: $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_{t+n} + \gamma^{n+1} V_\theta(s_{t+n+1})$. Similarly, a λ-*return* is a parametric mixture that can be recursively defined as $R_t = r_t + \gamma((1-\lambda)V_\theta(s_{t+1}) + \lambda R_{t+1})$, producing simple one-step returns for $\lambda = 0$ and Monte Carlo returns for $\lambda = 1$. When using a bootstrap, the update described above is said to perform *temporal-difference*, and in practice, the gradient through the *bootstrapping* function approximator V_θ is not taken to improve stability.

State-action value function The expected return that a policy π will generate from a given state s, when starting with a given action a, is called a state-action value function, or *Q-function*, and is defined as:

$$Q^\pi(s,a) = \mathbb{E}_{s'\sim p(.|s,a),a'\sim\pi(.|s')}\left[r(s,a,s') + \gamma Q^\pi(s',a')\right]$$

As for state value functions, to improve a parametric approximation Q_θ towards the true state value function Q^π of a policy π, estimates are used. Given a state s, action a, and return R, the loss is defined as:

$$L_\theta(s,a,R) = \frac{1}{2}\left(Q_\theta(s,a) - R\right)^2$$

and by taking its gradient with respect to the parameters θ, we get:

$$\nabla_\theta L_\theta(s,a,R) = \left(Q_\theta(s,a) - R\right)\nabla_\theta Q_\theta(s,a)$$

The return R is an approximation of the expected return from s when starting with action a and then following π. Once again, it can be constructed via dynamic programming or estimated via sampling and bootstrapping, by replacing V_θ by Q_θ. In effect, both types of value function are linked like this:

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

$$Q^\pi(s,a) = \mathbb{E}_{s'\sim p(.|s,a)}\left[r(s,a,s') + \gamma V^\pi(s')\right]$$

Sarsa In the Sarsa algorithm, the state-action value function Q^π of the policy π is approximated iteratively. This policy is the one used to interact with the environment and is built on top of the learned Q-values by mixing exploratory actions with greedy ones. For example, the ε-greedy exploration method selects the greedy action $\arg\max_a Q(s,a)$ with $(1-\varepsilon)$ probability and uniformly from $\mathcal{A}$ with probability ε. Boltzmann exploration, another exploration strategy, constructs a Boltzmann distribution using the Q-values, giving higher probability to higher value actions. For a given sample s, a, r, s', a' (giving its name to the method), the loss function is defined as:

$$L_\theta(s,a,r,s',a') = \frac{1}{2}\left(Q_\theta(s,a) - (r + \gamma Q(s',a'))\right)^2$$

and taking the gradient of this loss with respect to θ, we get:

$$\nabla_\theta L_\theta(s,a,r,s',a') = \left(Q_\theta(s,a) - (r + \gamma Q(s',a'))\right)\nabla_\theta Q_\theta(s,a)$$

Updating θ using this gradient improves the estimate of Q^π. Interestingly, since Q has changed, π has changed as well, but after some time, the policy and Q-function

will converge. In parallel, the amount of exploration can be reduced over time, to allow the policy to improve, letting the Q-function track the changes in π. Sarsa is called an *on-policy* algorithm because the data produced by a policy is valid only for the policy that generated it and must be discarded after an update.

2.1.4 Policy improvement

Q-learning In the Q-learning algorithm (Watkins & Dayan, 1992), the state-action value function Q^* of the optimal policy π^* is iteratively approximated with a learned Q-function Q. In contrast to Sarsa, transitions s, a, r, s' are provided, and the next a', used in the updates, does not come from the policy, but by taking the action maximizing the estimated Q-value in s'. The fact that the target value $r + \gamma \max_{a'} Q(s', a')$ is independent from the policy used in the environment, makes Q-learning an *off-policy* method and allows it to use transitions coming from past experience, or demonstrations. Given a Q-function parameterized by θ, the loss function is defined as:

$$L_\theta(s, a, r, s') = \frac{1}{2}\left(Q_\theta(s, a) - \left(r + \gamma \max_{a'} Q(s', a')\right)\right)^2$$

and taking the gradient of this loss with respect to θ, we get:

$$\nabla_\theta L_\theta(s, a, r, s') = \left(Q_\theta(s, a) - \left(r + \gamma \max_{a'} Q(s', a')\right)\right) \nabla_\theta Q_\theta(s, a)$$

Stochastic policy gradient Instead of defining a policy over value functions, the policy can directly be represented and optimized using a learned action distribution. The stochastic policy gradient methods (Sutton et al., 2000) rely on a parametric policy and the gradient of the expected return with respect to the policy parameters. For a policy parameterized by μ, an on-policy transition s, a, and on-policy return R from that step, the loss function L is defined as:

$$L_\mu(s, a, R) = -\log\left(\pi_\mu(a|s)\right) R$$

and taking the gradient of this loss with respect to μ, we get:

$$\nabla_\mu L_\mu(s, a, R) = -\nabla_\mu \log\left(\pi_\mu(a|s)\right) R$$

The return R is an approximation of the expected return from s by performing a and then following π. It can be a Monte Carlo return or use a learned value function $R = Q(s, a)$ or $R = r + \gamma V(s')$. Since these quantities can induce a lot of variance in the updates, it is usually beneficial to subtract a state-dependent quantity, called a *baseline*. In practice, it is common to estimate *advantages* using a n-step return

combining n perceived rewards, a bootstrap, and subtracting the value of the state s like so: $A_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^n r_{t+n} + \gamma^{n+1} V(s_{t+n+1}) - V(s_t)$. Methods using both an explicit policy and a value function are called *actor-critic* methods.

Deterministic policy gradient Stochastic policies are useful for various reasons and in particular in POMDPs or non-stationary environments such as adversarial environments. For example, when playing the game of "rock paper scissors" against a reasonably smart opponent, sticking to one of the three actions is not an optimal policy. However, in many scenarios, when action spaces are continuous, representing a direct continuous mapping from states to actions can be desirable. In particular, the deterministic policy gradient methods are an adaptation of Q-learning to continuous control and offer a very direct way of approximating π^*. For a deterministic policy parameterized by μ, a learned Q-function parameterized by θ, and an off-policy transition s, a, r, s', the loss of the Q-function is defined as:

$$L_\theta(s, a, r, s') = \frac{1}{2} \left(Q_\theta(s, a) - (r + \gamma Q(s', \pi_\mu(s')))\right)^2$$

and taking the gradient of this loss with respect to θ, we get:

$$\nabla_\theta L_\theta(s, a, r, s') = \left(Q_\theta(s, a) - (r + \gamma Q(s', \pi_\mu(s')))\right) \nabla_\theta Q_\theta(s, a)$$

The loss function of the policy only needs states and is defined as:

$$L_\mu(s) = -Q_\theta(s, \pi_\mu(s))$$

and taking the gradient of this loss with respect to μ, we get:

$$\nabla_\mu L_\mu(s) = -\nabla_a Q_\theta(s, a) \mid_{a=\pi_\mu(s)} \nabla_\mu \pi_\mu(s)$$

2.2 Deep learning

Deep learning is a set of methods to construct parametric differentiable computation graphs and train them from data using gradient-based optimization. In this section, we explain some of the key elements of the deep learning revolution.

2.2.1 Differentiable computational graphs

A deep learning model, also called an *artificial neural network* (ANN), is a network of parameterized functional modules, generally organized in layers, allowing computation to be carried out from the input layers to the output layers. The values stored in each layer are called units or neurons.

Universal approximation theorem The universal approximation theorem states that any continuous function can be arbitrarily well approximated by a neural network with at least one hidden layer by varying the number of units. Furthermore, increasing the number of hidden layers allows networks to create more complex hierarchical computations. Deeper networks tend to be more powerful but harder to train. The word "deep" in deep learning comes from this powerful depth dimension.

2.2.2 Optimization

Backpropagation Deep learning is based on gradient-based optimization and, in particular, gradient descent and variations of it. To perform gradient descent, a first forward pass through the network is performed with a batch of inputs. The outputs are generated and used to compute a loss. Then, the gradient of the loss, that is the list of the partial derivatives, has to be computed with respect to each of the network parameters. This is performed efficiently in one backward pass, using the backpropagation algorithm, or backprop, that applies the chain rule, computing the gradient one layer at a time, iterating backward from the last one.

Automatic differentiation Importantly, for the chain rule to be used, the derivative function of each layer must be known, and earlier implementations of the backpropagation algorithm required each of these to be specified by the user. Fortunately, in all modern deep learning frameworks, automatic differentiation, also known as autograd, automatically determines those derivatives by recognizing the primary computation blocks used such as sums, products, exponentials, powers or sine functions and using their known derivatives.

Optimizers Vanilla gradient descent can be augmented to speed up convergence. For example, momentum creates updates by linearly combining the current gradient with the previous update. The inertia in the resulting updates allow them to overcome the oscillations of noisy gradients and escape flat surfaces of the search space. Other methods, based for example on second order derivatives, can also be used to create more informed updates. Some optimization techniques require a fair amount of memory and compute, creating a trade-off between sample efficiency and clock time. Notable examples of optimizers are AdaGrad (Duchi et al., 2011), RMSProp (Tieleman & Hinton, 2012), and Adam (Kingma & Ba, 2014).

Batches In theory, given a sufficiently small learning rate, gradient descent reduces the value of a loss function when the gradient is computed over the entire dataset. However, in practice, a dataset is rarely used entirely for every training step. An

obvious reason is that this is just not practical in most cases because the amount of memory required is too large. Feeding an entire dataset containing potentially millions of examples, and computing the neural activations for each example in parallel and performing backpropagation would just take more memory than is available on most machines. In practice, stochastic gradient descent is usually chosen. It relies on iterative gradient descent steps using smaller batches of data, approximating the full batch gradient descent. It is also worth noting that batches of inputs generally allow for faster inference or backprop per input than for individual calls due to the parallelization capabilities of hardware and software.

2.2.3 Layers

Layers are the main building blocks of neural networks. Stacking them increases the depth and therefore expressiveness of networks.

Skip connections Deep neural networks tend to suffer from the *vanishing gradient* problem, where gradients become increasingly small with depth. This is emphasized by some activation functions creating a squashing of the values, reducing gradients. Some strategies exist to overcome this issue, such as skip connections, where the result of previous layers is reused at various stages. The most common type of network using skip connections is the residual neural network or *ResNet*, in which the result of a block of layers is added to the input of the same block, to allow information to be maintained and enable shorter paths for gradients to flow through.

Activation functions Most existing layers use simple linear transformations of their inputs. To build powerful functions, nonlinearities have to be used after such layers. It is the alternation of linear parametric operations and nonlinear (often nonparametric) transformations that gives deep neural networks their expressiveness beyond linear transformations. The most popular activation functions are the ReLU (rectified linear unit), ELU (exponential linear unit), sigmoig (logistic), tanh (hyperbolic tangent), and softplus.

Fully connected layers The simplest type of layer is the fully connected layer, also called dense, or perceptron. It requires one hyperparameter: the number of output units. It contains two parameters: a weight matrix W and a bias vector b. Given an input tensor X, and an activation function f, the output Y is generated like so: $Y = f(W.X + b)$. A *multilayer perceptron* (MLP) is a stack of fully connected layers.

Convolutional layers A 2D convolutional layer is loosely inspired by neurons in the visual cortex. It applies a kernel with a receptive field over a 2D feature map. A 2D feature map is a tensor with width, height, and depth. The depth number is also called the number of channels. Grayscale images are feature maps with one channel, while color RGB images are feature maps with three channels for red, green, and blue. Convolutions are defined by a kernel with width, height, and depth that is moved over the input feature map to create an output feature map. The dimensions of this kernel define its receptive field, and by applying the same kernel at various places over the input feature map, the resulting pattern matching is translation invariant. Applying the kernel to a patch of the input feature map effectively multiplies this input tensor with a tensor weight and adds a tensor bias, similar to a fully connected layer in three dimensions. A 2D convolutional layer requires a few hyper parameters such as: the number of output channels, the kernel size that specifies the 2D dimensions of the receptive field of the kernel, the strides that give the 2D offset in number of pixels between two consecutive applications of the kernel, the padding that tells the number of pixels added to both borders of the input feature map, and the dilation rate that allows to increase the size of the receptive field by inserting holes between the kernel elements (inflation). Furthermore, downsampling can also be performed using a pooling layer that aggregates neighboring values, for example, with a max operator, to reduce the size of the feature maps. A convolutional neural network, often abbreviated *ConvNet* or CNN, is a stack of convolutional layers generally followed by a flattening operation and an MLP.

Recurrent layers To allow neural networks to process sequences of input data and retain information, recurrent layers have been introduced. They all rely on some internal state being carried over between unroll steps, and mostly have one hyperparameter specifying the size of the generated output. The simplest form of recurrent neural network (RNN) takes in input both the current input and the previous output, concatenates them, and generates a new output using an internal dense layer. This output is then used itself with the next input to generate the next output, and so on. The sequence of inputs and outputs can be used in various ways depending if the task at hand is many to one (only the last output is sent to the next layer), one to many (the same input is repeated multiple times in input, and the many outputs are sent to the next layer), or many to many (at every unroll step, a different input is provided, and the generated output is sent to the next layer). In any case, the last output generated by a simple RNN is the result of several passes through the same dense layer. This makes the retention of information increasingly harder for the network, and the unroll of a recurrent layer is effectively the same

as a deep network with a repeated layer, creating issues with vanishing gradients once again. To overcome this issue, *long short-term memory* (LSTM) (Hochreiter & Schmidhuber, 1997) and gated recurrent unit (GRU) (Cho et al., 2014) layers have been proposed. They rely on gating mechanisms involving multiple internal dense layers controlling the amount of forgetting and addition performed at every unroll step, drastically increasing memory capacities and limiting vanishing gradient issues.

Attention layers In the deep learning literature, the word "attention" has been used to refer to mechanisms allowing parts of a large amount of input to be emphasized. Currently, the most common type of attention, used by the popular Transformer network (Vaswani et al., 2017), relies on queries, keys, and values. On one side, an unordered set of vectors is projected into key and value vectors using linear layers, while on the other side, a possibly different set of unordered vectors is projected into query vectors. The dot product between keys and queries generates scalars that after exponentiation and normalization provide coefficient weights. These weights are applied to the corresponding value vectors in a weighted sum to generate a new set of values in which the new vectors aggregate information coming from different parts of the input sets. Transformers were originally applied to translation tasks, in which the first input set was representing the text in the source language, and the second set was representing the text translated so far in the target language. Paying attention to both input sequences provides a very effective way of performing the translation without the need for memorization. Attention directly attends to parts of the inputs without having to first encode those sequences iteratively with recurrent networks. This approach represents a true paradigm shift in deep learning, as those layers produce "soft weights" that change with the inputs and can handle dynamic set sizes.

2.2.4 Probability distributions

Probability distributions are used at the end of a large number of neural networks. They are used, for example, in classifiers, image or text generators, and policies.

Logistic function The simplest type of probability distribution is defined using the logistic function. The probability of one outcome is modeled using a single scalar x, also called "logit", that is passed through a sigmoid function $1/(1 + \exp(-x))$. A probability of 0 is represented by a large negative logit, while a probability of 1 is represented by a large positive one.

Softmax The generalization of a logistic function to multiple classes is called a softmax. The probability of classes is modeled using a logit vector x that is passed through a normalized exponential function. The probability of a class i is $\exp(x_i)/\sum_j \exp(x_j)$.

Gaussian For continuous spaces, a multivariate Gaussian distribution, also called a multivariate normal distribution, can be used. It is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions. Its probability density function has a characteristic bell curve centered around its mean and with a covariance matrix that defines how the distribution spreads over its dimensions. The covariance matrix must be positive definite which is not straightforward to represent by a neural network. For simplicity, most multivariate normal distributions are therefore built to be diagonal, meaning that the covariance matrix is zero everywhere except on the main diagonal (scale). This makes each dimension independent. Therefore a *diagonal Gaussian* is represented by two vectors: one for the mean and one for the scale. A particular case of such distribution, when the scale vector is filled with ones, is called a standard normal distribution.

Reparameterization trick If a probability distribution is used somewhere in the middle of a neural network, the generated stochastic samples do not directly allow gradients to flow through. To remediate to this issue, the reparameterization trick can be used. It relies on rewriting the generation of the sample as a differentiable function of deterministic values produced by the network and a random sample generated by an external distribution of probabilities. For example, a mean vector μ and a scale vector σ generated by a neural network can be combined with a random noise vector ϵ, generated by a multivariate standard normal distribution, to create a sample from a diagonal Gaussian distribution. This sample is $y = \mu + \sigma\epsilon$ and allows gradients to flow through μ and σ.

2.3 Deep reinforcement learning

2.3.1 Agents

A large number of agents have been proposed in the deep reinforcement learning literature. The agents used in the rest of this thesis are summarized below.

DQN: Deep Q-Network DQN (Mnih et al., 2013, 2015) is the first modern deep reinforcement learning agent. It was the first one to demonstrate such high performance on a large number of Atari 2600 games from pixel inputs. It is based

on the Q-learning algorithm, with an online ConvNet representing the Q-function, a target network used for bootstrapping that consists of an older version of the online network, and a replay buffer for replaying previous transitions during training. This agent is used in Chapter 4 and Chapter 5.

A2C: Advantage Actor Critic A2C (Mnih et al., 2016), also called "Vanilla Policy Gradient" (VPG), can use advantages from λ-returns and a learned value function to update a stochastic policy via policy gradient (Sutton et al., 2000; Schulman et al., 2015b; Mnih et al., 2016). It is fairly straightforward to implement as it only requires a stochastic policy and a state value function, but suffers from poor sample efficiency. In effect, its on-policy nature does not allow the collected data to be used more than once. To compensate for this, distributed training is often used with this agent. This technique uses several parallel environments, synchronous, or asynchronous as democratized by the A3C agent. This agent is used in Chapter 12.

TRPO: Trust Region Policy Optimization TRPO (Schulman et al., 2015a) uses a conjugate gradient optimizer to take a large update step of policy gradient while satisfying a Kullback–Leibler (KL) divergence constraint between the policy before and after performing an update. This agent is used in Chapter 12.

PPO: Proximal Policy Optimization PPO (Schulman et al., 2017) approximates the trust region optimization method proposed by TRPO, while remaining much simpler to implement as a variant of A2C. A single batch can be used multiple times with a traditional optimizer thanks to a loss based on clipped ratios, limiting how much a policy can diverge from the one that generated the data. This agent is used in Chapter 10 and Chapter 12.

DDPG: Deep Deterministic Policy Gradient DDPG (Lillicrap et al., 2015) uses a deterministic actor trained via deterministic policy gradient (Silver et al., 2014). The critic network is a state-action value function allowing the output of the actor to be directly evaluate by the critic in a differentiable way. The actor and critic networks are both updated off-policy and target versions of those are used to stabilize training as for DQN. This agent is used in Chapter 4 and Chapter 12.

SVG(0): Stochastic Value Gradient SVG(0) (Heess et al., 2015) can be viewed as a stochastic variant of DDPG. During off-policy training, multiple actions are sampled and evaluated to form estimates of state values in a differentiable way thanks to the reparametrization trick. This agent is used in Chapter 7.

D4PG: Distributed Distributional DDPG D4PG (Barth-Maron et al., 2018) is a variant of DDPG, making use of a distributional critic (Bellemare et al., 2017), n-step returns, and a prioritized experience replay (Schaul et al., 2015b). To be more accurate, D4PG also uses distributed training, but this training element can be viewed as an infrastructure detail that is independent of the agent description. Nevertheless, since the name D3PG is not commonly used, D4PG will still be employed in this thesis, regardless of the distributed aspect of the training. This agent is used in Chapter 8 and Chapter 12.

TD3: Twin Delayed DDPG TD3 (Fujimoto et al., 2018) is a variant of DDPG, making use of a pair of critics, action noise in the target actor, and delayed policy updates. These are inspired by double Q-learning (Van Hasselt, 2010) and connections between target networks and overestimation bias. These changes reduce the function approximation error found in DDPG that leads to overestimated values and suboptimal policies. This agent is used in Chapter 8 and Chapter 12.

SAC: Soft Actor-Critic SAC (Haarnoja et al., 2018b) is similar to SVG(0) with a few more tricks. It uses a pair of critics as in TD3, an entropy-based reward augmentation to encourage diversity and help exploration, and a squashed Gaussian policy to keep the stochastic distribution bounded. This agent is used in Chapter 12.

MPO: Maximum a Posteriori Policy Optimisation MPO (Abdolmaleki et al., 2018) uses a complex relative-entropy objective taking advantage of the duality between control and estimation. As in SVG(0) and SAC, a stochastic policy and a Q-function are used, but the optimization of the actor uses a parametric dual loss. This combination can be particularly powerful if carefully tuned, but its complexity makes its use rather rare in the literature. It is nevertheless used in Chapter 7 and Chapter 12.

2.3.2 DQN improvements

Double Q-learning Both tabular Q-learning and DQN have been shown to suffer from the overestimation of the action values (Van Hasselt, 2010; Van Hasselt et al., 2016). This overoptimism stems from the fact that the same values are accessed in order to both select and evaluate actions. In the standard DQN algorithm, a copy of the Q-network from several steps in the past, called the target network, is used to select the next greedy Q-value involved in the Q-learning updates. To address the overoptimism in the Q-value estimations, Van Hasselt et al. (2016) propose the

Double DQN (DDQN) algorithm that uses the current Q-network to select the next greedy action, but evaluates it using the target network.

Dueling networks The dueling network architecture (Wang et al., 2016) explicitly separates the representation of the state value and the (state-dependent) action advantages into two separate branches while sharing a common feature-learning module among them. The two branches are combined, via a special aggregation layer, to produce an estimate of the state-action value function. By training this network with no additional considerations than those used for the DQN algorithm, the dueling network automatically produces separate estimates of the state value and advantage functions. Wang et al. (2016) introduce multiple aggregation methods to combine the state value and advantages. They demonstrate that subtracting the mean of the advantages from each individual advantage and then summing them with the state value, results in improved learning stability when compared to the naive summation of the state value and advantages. The dueling network architecture has been shown to lead to better policy evaluation in the presence of many similar-valued (or redundant) actions, and thus achieves faster generalization over large action spaces.

Distributional value functions Value functions are defined as expectations due to stochasticity arising from both the environment's transitions and the agent's policy. Traditionally, the function approximators used to represent value functions output one scalar per value, and by successively training those outputs to match sampled target values, an average is in effect learned. However, a more robust approach called distributional learning, consists in approximating the distribution of possible values and has been shown to improve the performance and stability in various situations. For example, if only two returns with equal probability are possible, the expected value is the average of those two values, and training a network with samples from one or the other outcomes creates large errors in both directions. On the contrary, a distribution giving equal probability to both values and zero probability to others can easily converge.

Prioritized experience replay Experience replays enable off-policy algorithms to reuse past experiences or demonstrations. With the simple experience replay used by DQN, past transitions are sampled uniformly regardless of their significance. To learn more efficiently, Schaul et al. (2015b) propose to prioritize the replay of transitions with higher expected learning progress, as measured by the magnitude of the learning error they generate.

Part I

Designing agents

Chapter 3

Agent structure

Figure 3.1: Radial layout enumerating a number of elements involved in the design of agents. These include exploration strategies, update rules, experience replays, neural networks, environment models, and hierarchies.

3.1 The agent

In reinforcement learning, an agent is an algorithm that receives inputs from an environment, typically observations, rewards and terminations, and produces actions that influence the environment. It is, for example, the program inside a robot, but it does not include the body of the robot. Anything else than the agent is part of the environment. Various components must be combined to build agents. Some of the main groups are shown in Figure 3.1 and are described in the following sections. Of course, the delimitation of an agent into components is not always clear, and the proposed grouping is only valid to some extent. For example, while exploration can be treated as a specific process that generates exploratory actions with some probability, it could also be discovered through learning if the agent can observe the effect of its actions. In this case exploration becomes part of the policy itself. It is worth noting that some components are not compatible with others. For example, hindsight experience replay (HER) (Andrychowicz et al., 2017) was designed to be used with a goal-conditioned policy and an off-policy update strategy. Nevertheless, when possible, agents should remain modular to allow for the composition of ideas.

3.2 Agent components

Exploration strategies The objective of the agent is to maximize its return, the discounted sum of future rewards. An agent typically starts with no particular knowledge and has to try various actions in different situations to *explore*, gathering data useful to learn its policy. While the policy becomes more accurate and optimal, the agent can *exploit* the best known actions to maximize its return. If an agent is evaluated throughout its lifetime, it should efficiently balance exploration and exploitation over the course of its training. A large number of exploration strategies have been proposed. For example, epsilon gray, upper confidence bounds, Boltzmann exploration, and Thompson sampling have been thoroughly studied theoretically in controlled scenarios such as bandit problems. To address more challenging environments, including sparse rewards or no reward at all, more sophisticated exploration strategies have been later proposed, based, for example, on randomized value functions (Osband et al., 2016; Fortunato et al., 2017; Plappert et al., 2017; Osband et al., 2018), unsupervised policy learning (Gregor et al., 2016; Achiam et al., 2018; Eysenbach et al., 2018), intrinsic motivation (Schmidhuber, 1991; Oudeyer et al., 2007; Barto, 2013; Bellemare et al., 2016; Ostrovski et al., 2017; Tang et al., 2017; Pathak et al., 2017; Burda et al., 2018b,a; Badia et al., 2020), and quality diversity (Cully et al., 2015; Pierrot et al., 2020; Ecoffet et al., 2021).

Update rules Learning is essential to intelligence. It allows an agent to improve and adapt. Update rules have therefore been at the center of most of the research in reinforcement learning. From tabular to parametric functions, a large number of rules have been proposed to update value functions and policies. These updates generally fall into two categories. On the one hand, on-policy approaches such as Sarsa (Rummery & Niranjan, 1994) and stochastic policy gradient (Sutton et al., 2000) only consider data coming from the behavior policy. They cannot directly reuse past data or demonstrations without having to account for off-policyness via importance sampling (Munos et al., 2016; Espeholt et al., 2018). On the other hand, off-policy approaches such as Q-learning (Watkins & Dayan, 1992) and deterministic policy gradient (Silver et al., 2014) can reuse past data and demonstrations directly and are therefore typically more data efficient. However, a number of approaches based on trust regions (Schulman et al., 2015a, 2017; Abdolmaleki et al., 2018) have been recently proposed to allow more data efficient updates while remaining stable.

Experience replays Transitions generated by an agent's interaction with its environment can generally be used in one of two ways. In online methods, newly collected samples are used directly and discarded. This is typically the case with on-policy methods, but off-policy methods can also be used in an online manner to focus learning on recent experience. Although tabular methods can benefit directly from online learning, neural networks tend to suffer from forgetting when the data distribution is not kept approximately uniform. To address this problem, offline learning is generally preferred because it relies on storing transitions in a buffer and replaying them in a more uniform manner. The standard experience replay (Lin, 1992a; Mnih et al., 2015) samples past transitions uniformly to minimize correlation between successive batches. Relatively few improvements have been added to this rather simple approach, other than prioritizing transitions with a higher expected learning signal (Schaul et al., 2015b). However, it should be noted that implementing efficient replays is more difficult than it seems (Kapturowski et al., 2018; Cassirer et al., 2021). Indeed, many agents require preprocessing steps such as frame stacking and sampling of consecutive transitions to compute n-step returns or to feed recurrent networks. To add to the challenge, multiple streams of experiments can be collected simultaneously when distributed training is used. Nonetheless, a few other types of replay exist, including hindsight experience replay (Andrychowicz et al., 2017) that generates goal-directed counterfactual transitions and episodic memories (Blundell et al., 2016; Pritzel et al., 2017) that directly intertwine representations and recall.

Neural networks While tabular representations have been used extensively in traditional reinforcement learning research, the deep learning revolution has made the use of powerful parametric function approximations indispensable nowadays. DQN (Mnih et al., 2013, 2015) used a simple stack of convolutions and fully connected layers to output the Q-values of every action given a stack of frames in input. A large number of architectures have been proposed since then, including the use of recurrent networks (Espeholt et al., 2018; Kapturowski et al., 2018), dueling (Wang et al., 2016) and distributional value functions (Bellemare et al., 2017), stochastic and deterministic policies, and more recently attention-based mechanisms (Parisotto et al., 2020).

Environment models To increase the sample efficiency of an agent or to enable planning, a model of the environment can be a very useful tool for an agent. This model can be given, as is the case when the rules of a board game are known, or it can be learned, including transition, reward, and termination functions. Agents can then be trained using both transitions from the environment and from imagined transitions (Sutton, 1990, 1991; Ha & Schmidhuber, 2018; Hafner et al., 2019a, 2020). Planning can also be performed using rollouts (Hafner et al., 2019b), as in model predictive control, or tree search. In particular, Monte Carlo tree search (MCTS) has been used extensively in AlphaGo (Silver et al., 2016), AlphaGo Zero (Silver et al., 2017b), AlphaZero (Silver et al., 2017a), and MuZero (Schrittwieser et al., 2020) to achieve superhuman performance in board games and other domains. Features from imagined rollouts can also be extracted and combined to the current state to improve the performance on planning tasks (Racanière et al., 2017). Finally, learning to model the environment can be used as an auxiliary task to help useful representations emerge in networks (Hessel et al., 2021).

Hierarchies In theory, in fully observable Markovian environments, learning "reactive" or "shallow" policies that directly relate observations to action probabilities should be possible. However, enforcing a stronger hierarchy in how actions are perceived may have a number of advantages, including temporal abstraction (Dayan & Hinton, 1992; Kaelbling, 1993a; Sutton, 1995; Parr & Russell, 1997; Wiering & Schmidhuber, 1997; Precup et al., 1998; Dietterich, 2000; Vezhnevets et al., 2016). The option framework (Sutton et al., 1999; Precup, 2000) is the main theoretical framework for temporal abstraction. It relies on the subdivision of a policy into specialized sub-policies, to facilitate learning, planning, and exploration. Indeed, manipulating macro decisions such as "go to the airport", "take a plane", and "attend a conference" is simpler to process than the vast number of "raw" actions that

constitute the trip. Learning options and policies over options (Stolle & Precup, 2002; Bacon et al., 2017; Smith et al., 2018; Barreto et al., 2021) have been a key challenge to the successful application of the framework. Another approach to enforcing hierarchy is to use goals to be satisfied, which could be abstract representations or concrete states (Vezhnevets et al., 2017; Nachum et al., 2018a,b; Lynch et al., 2020). In addition, approaches combining task-specific high-level controllers and task-agnostic low-level controllers in the context of multitasking or skill reuse have proven particularly effective (Haarnoja et al., 2018a; Eysenbach et al., 2018; Merel et al., 2018b).

3.3 Elements of agent design in the thesis

1. In Chapter 4, a novel method called action branching is proposed to apply discrete action algorithms to multidimensional action spaces. In particular, a new agent called BDQ is presented, containing a number of improvements adapted from DQN-based agents, and is evaluated on discretized continuous control tasks.

2. In Chapter 5, a novel method called Q-map is proposed to scale all-goal updates in goal-oriented reinforcement learning. In particular, the proposed approach makes use of convolutional networks and produces Q-values for goal coordinates from pixels and updates them at once. Q-map is then used for a novel exploration method based on trajectories towards random goals.

3. In Chapter 7, motor skills are built both from motion capture tracking and complementary tasks and then reused through a hierarchical architecture with a variational embedding space. Several aspects are studied including latent space dimensionality and regularization, type of policy heads, and network structure.

4. In Chapter 8, continuous control of muscles is studied. This unusual action space is large and over-actuated with non-linear dynamics. While this work focuses more on the design of the environment than the agent, the results obtained with a carefully tuned state-of-the-art agent demonstrate the difficulty of the exploration problem in this type of environment. Finding efficient and natural motion require a combination of motion capture tracking and correlated exploration noise.

5. In Chapter 10, the difference between time limits used in time-limited tasks and the ones used in time-unlimited tasks is discussed. Several examples are used to demonstrate the fact that a notion of time needs to be tracked by the agent in the first case and that bootstrapping is required at the end of partial-episodes when training in the second case.

6. In Chapter 11, a deep learning called Ivy is proposed to unify popular deep learning frameworks. The choice of a deep learning framework is inevitable when designing agents and Ivy proposes an elegant solution for inter-framework portability.

7. In Chapter 12, a novel deep reinforcement learning library called Tonic is proposed. It is designed for fast prototyping, using a modular approach, and containing a number of important continuous-control agents from classic to state of the art, benchmarked on a large number of popular environments.

Chapter 4

Learning to control large multidimensional action spaces

This chapter is based on the paper "Action branching architectures for deep reinforcement learning" (Tavakoli et al., 2018) published at the AAAI 2018 conference. A previous version was first presented at the NIPS 2017 deep RL symposium.

The source code is available at:
https://github.com/atavakol/action-branching-agents.

The main research questions are:

- How to transform continuous action spaces into discrete ones?
- How to apply DQN to the resulting large multidimensional action spaces?

4.1 Introduction

As discussed in Subsection 2.3.1, DQN Mnih et al. (2015) is a key deep reinforcement learning agent that has introduced several standard elements for agent design, including a deep function approximation, a target network, an experience replay, and a large-scale benchmark. It has been extensively studied and has benefited from several improvements over the years (Hessel et al., 2018), including double Q-learning (Van Hasselt, 2010), prioritized experience replay (Schaul et al., 2015b), and dueling networks (Wang et al., 2016) described in Subsection 2.3.2.

At the heart of DQN is the Q-learning algorithm (Watkins & Dayan, 1992), that requires to predict the Q-value of every possible action. Given this limitation, it has traditionally been used with discrete action spaces of moderate sizes. However, one might wish to apply DQN and its improvements to continuous action spaces. An approach could be to first transform the action space by discretizing every action dimension and then enumerate the resulting set of tuples. However, such an approach

suffers from the rapid growth of the Cartesian product of sub-actions. Formally, for an environment with an N-dimensional action space and n_d discrete sub-actions for each dimension d, a total of $\prod_{d=1}^{N} n_d$ possible actions needs to be considered.

In this work, we propose to allow the DQN agent to predict the value of a given action tuple without having to enumerate all of them. To do so, we propose to use what we call "action branching", where the values for every dimension are predicted semi-independently. In practice, a shared encoder module is followed by a number of parallel action branches, one per action dimension, outputting Q-values simultaneously for each sub-action in the corresponding action dimension. This architectural choice, allows a much more gentle growth of the number of outputs with the number of action dimensions. Formally, a total of $\sum_{d=1}^{N} n_d$ values must be predicted. If the number of sub-actions n is equal between all the dimension, this is in fact a linear growth Nn.

To demonstrate the effectiveness of the approach, we propose a novel agent, called Branching Dueling Q-Network (BDQ), which is a branching variant of the Dueling Double DQN (Wang et al., 2016). We evaluate BDQ on a variety of complex control problems via fine-grained discretization of the continuous action space. Our empirical study shows that BDQ can scale robustly to environments with high-dimensional action spaces and even outperform the Deep Deterministic Policy Gradient (DDPG) agent (Lillicrap et al., 2015) in the most challenging task with a corresponding discretized combinatorial action space of approximately 6.5×10^{25} action tuples. To the best of our knowledge, this work is the first to apply discrete-action algorithms to environments with discrete action spaces of this magnitude.

Furthermore, in order to demonstrate the vital role of the shared module in our architecture, we compare BDQ against a completely independent variant, which we refer to as Independent Dueling Q-Network (IDQ), an agent consisting of multiple independent networks, one for each action dimension, and without any shared parameters among the networks. The results show that the performance of IDQ quickly deteriorates with increasing action dimensionality, suggesting that the shared module is essential to the stability of BDQ.

Finally, we wish to mention that action branching can be seen as a partial distribution of control, somewhat similar to the one found in octopuses. Indeed, they possess a complex neural systems where each arm is able to function with a degree of autonomy and even respond to stimuli after being detached from the central control. In fact, more than half of the neurons in an octopus are spread throughout its body, especially within its arms (Godfrey-Smith, 2016). Since octopuses' arms

have virtually unlimited degrees of freedom, they are highly difficult to control in comparison to jointed limbs. This calls for the partial delegation of control to the arms to work out the details of their motions themselves. Interestingly, arms not only have a degree of autonomy, they have also been observed to engage in independent exploration (Godfrey-Smith, 2016).

4.2 Related work

4.2.1 Continuous control applied to discrete actions

To enable the application of reinforcement learning algorithms to large-scale, discrete-action problems, Dulac-Arnold et al. (2015) propose the Wolpertinger policy architecture based on a combination of DDPG and an approximate nearest-neighbor method. This approach leverages prior information about the discrete actions in order to embed them in a continuous space upon which it can generalize, meanwhile, achieving logarithmic-time lookup complexity relative to the number of actions. Due to the underlying algorithm being essentially a continuous-action algorithm, this approach may be unsuitable for environments with naturally discrete action spaces where no assumption should be imposed on having associated continuous space correlations. Also, this approach does not enable the application of discrete-action algorithms to environments with high-dimensional action spaces as it relies on a continuous-action algorithm.

4.2.2 Autoregressive action selection

Concurrent to our work, Metz et al. (2017) have developed an approach that can deal with environments with high-dimensional discrete action spaces using Q-learning. They use an autoregressive network architecture to sequentially predict the action value for each action dimension. This requires manual ordering of the action dimensions which imposes a priori assumptions on the structure of the task. Additionally, due to the sequential structure of the network, as the number of action dimensions increases, so does the noise in the Q-value estimations. Therefore, with increasing number of action dimensions, the Q-value estimates on the latter layers may become too noisy to be useful. Due to the parallel representation of the action values or policies, our proposed approach is not prone to cumulative estimation noise with increasing number of action dimensions and does not impose manual action factorization. Furthermore, our proposed approach is simpler to implement, as it does not require advanced neural network architectures.

4.2.3 Cooperative multi-agent approaches

A potential approach towards achieving scalability with increasing number of action dimensions is to extend deep reinforcement learning algorithms to fully-cooperative multi-agent settings in which each agent, responsible for controlling an individual degree of freedom, observes the global state, selects an individual action, and receives a team reward common to all agents. Tampuu et al. (2017) combine DQN with independent Q-learning, in which each agent independently and simultaneously learns its own state-action value function. Although this approach has been successfully applied in practice to environments with two agents, in principle, it can lead to convergence problems (Matignon et al., 2012). In this work, we empirically investigate this scenario and show that by maintaining a shared set of parameters among the action branches, our proposed approach is able to scale to high-dimensional action spaces.

4.3 Methods

4.3.1 Action branching architectures

The key insight behind the proposed action branching architecture is that, for solving problems in multidimensional action spaces, it is possible to optimize for each action dimension with a degree of independence. If executed appropriately, this altered perspective has the potential to trigger a dramatic reduction in the number of required network outputs. However, it is well known that the naive distribution of the value function or the representation of the policy among several independent function approximators is subject to numerous challenges that can lead to convergence problems (Matignon et al., 2012). To address this, the proposed neural architecture distributes the representation of the value function or the policy across several network branches, while maintaining a shared module among them to encode a latent representation of the common input state (see Figure 4.1). We hypothesize that this shared network module, paired with an appropriate training procedure, can play a significant role in coordinating the sub-actions that are based on the semi-independent branches and, therefore, achieve training stability and convergence to good policies. We believe this is due to the rich features in the shared network module that is trained via the backpropagation of the gradients originating from all the branches.

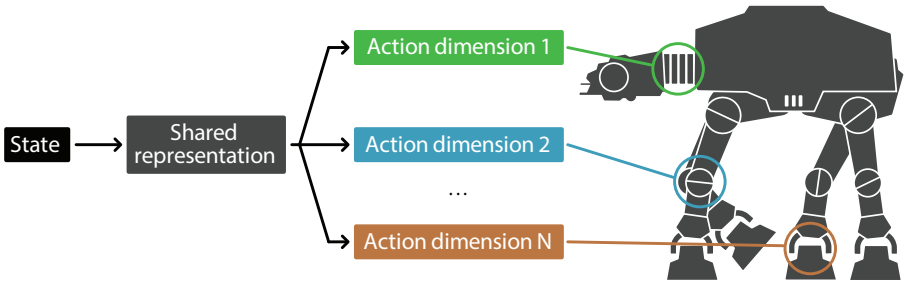


Figure 4.1: A conceptual illustration of the proposed action branching network architecture. The shared network module computes a latent representation of the input state that is then passed forward to the several action branches. Each action branch is responsible for controlling an individual degree of freedom, and the concatenation of the selected sub-actions results in a joint-action tuple.

4.3.2 Branching dueling Q-network

To verify this capability, we present a novel agent that is based on the incorporation of the proposed action branching architecture into a popular discrete-action reinforcement learning agent, the Dueling Double DQN (Dueling DDQN) (Wang et al., 2016). The proposed agent, which we call Branching Dueling Q-Network (BDQ), is only an example of how we envision our action branching architecture can be combined with a discrete-action algorithm in order to enable its direct application to tasks with high-dimensional, discrete, or continuous action spaces. We select DQN (Mnih et al., 2015) as the algorithmic basis for our proof-of-concept agent as it is a simple, yet powerful, off-policy algorithm with an excellent track record and numerous extensions (Hessel et al., 2018).

Although our experiments focus on a specific algorithm, we believe that the empirical verification of the aforementioned hypothesis suggests the potential of the proposed approach to enable the direct application of a spectrum of existing discrete-action algorithms to environments with high-dimensional action spaces.

Dueling architecture As shown in the action branching network of Figure 4.2, BDQ uses a common state-value estimator for all action branches. This approach, which can be thought of as an adaptation of the dueling network into the action branching architecture, generally yields a better performance. The use of the dueling architecture with action branching is particularly an interesting augmentation for learning in large action spaces. This is due to the fact that the dueling architecture can more rapidly identify action redundancies and generalize more efficiently by learning a general value that is shared across many similar actions. In order to adapt the dueling architecture into our action branching network, we distribute

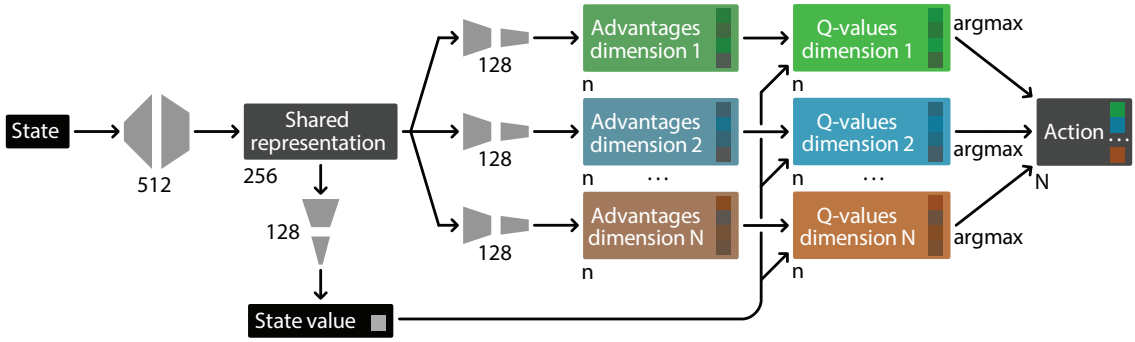


Figure 4.2: A visualization of the specific action branching network implemented for the proposed BDQ agent. When a state is provided at the input, the shared module computes a latent representation that is then used for evaluation of the state value and the factorized (state-dependent) action advantages on the subsequent independent branches. The state value and the factorized advantages are then combined, via a special aggregation layer, to output the Q-values for each action dimension. These factorized Q-values are then queried for the generation of a joint-action tuple. The weights of the fully connected neural layers are denoted by the gray trapezoids, and the size of each layer (i.e. number of units) is indicated.

the representation of the (state-dependent) action advantages on the several action branches, meanwhile, adding a single additional branch for estimating the state-value function. Similar to the dueling architecture, the advantages and the state value are combined, via a special aggregation layer, to produce estimates of the distributed action values. We experimented with several aggregation methods, and our best performing method is to locally subtract each branch's mean advantage from its sub-action advantages, prior to their summation with the state value. Formally, for an action dimension $d \in \{1, ..., N\}$ with $|\mathcal{A}_d| = n$ discrete sub-actions, the individual branch's Q-value at state $s \in \mathcal{S}$ and sub-action $a_d \in \mathcal{A}_d$ is expressed in terms of the common state value $V(s)$ and the corresponding (state-dependent) sub-action advantage $A_d(s, a_d)$ by:

$$Q_d(s, a_d) = V(s) + \big(A_d(s, a_d) - \frac{1}{n} \sum_{a'_d \in \mathcal{A}_d} A_d(s, a'_d)\big)$$

This aggregation method does not resolve the lack of identifiability for which the maximum and average reduction methods were originally proposed (Wang et al., 2016). However, based on our experimentation, this aggregation method yields a better performance than the naive alternative:

$$Q_d(s, a_d) = V(s) + A_d(s, a_d)$$

and the local maximum reduction method, which replaces the averaging operator

with a maximum operator:

$$Q_d(s, a_d) = V(s) + \big(A_d(s, a_d) - \max_{a'_d \in \mathcal{A}_d} A_d(s, a'_d)\big)$$

Temporal-difference target We tried several different methods for generating the temporal-difference (TD) targets for the DQN updates. A simple approach is to calculate a TD target, similar to that in DDQN, for each individual action dimension separately:

$$y_d = r + \gamma Q_d^-(s', \arg\max_{a'_d \in \mathcal{A}_d} Q_d(s', a'_d))$$

with Q_d^- denoting the branch d of the target network Q^-.

Alternatively, the maximum DDQN-based TD target over the action branches may be set as a single global learning target for all action dimensions:

$$y = r + \gamma \max_d Q_d^-(s', \arg\max_{a'_d \in \mathcal{A}_d} Q_d(s', a'_d))$$

The best-performing method, also used for BDQ, replaces the maximum operator with a mean operator:

$$y = r + \gamma \frac{1}{N} \sum_d Q_d^-(s', \arg\max_{a'_d \in \mathcal{A}_d} Q_d(s', a'_d))$$

Loss function There exist numerous ways by which the distributed TD errors across the branches can be aggregated to specify a loss. A simple approach is to define the loss as the expected value of a function of the averaged TD errors across the branches. However, because of the signs of such errors, their summation is subject to cancelation, which, in effect, generally reduces the magnitude of the loss. To overcome this, the loss can be specified as the expected value of a function of the averaged absolute TD errors across the branches. In practice, we found that defining the loss to be the expected value of the mean squared TD error across the branches mildly enhances the performance:

$$L = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\frac{1}{N} \sum_d \big(y_d - Q_d(s, a_d)\big)^2 \right]$$

where $\mathcal{D}$ denotes a (prioritized) experience replay buffer and a denotes the joint-action tuple $(a_1, a_2, ..., a_N)$.

Gradient rescaling During the backward pass, since all branches backpropagate gradients through the shared network module, we rescale the combined gradient before entering the deepest layer in the shared network module by $1/(N + 1)$.

Error for experience prioritization Adapting the prioritized replay into the action branching architecture requires an appropriate method for aggregating the distributed TD errors (of a single transition) into a unified one. This error is then used by the replay to calculate the transition's priority. In order to preserve the magnitudes of the errors, for BDQ, we specify the prioritization error to be the sum of the absolute TD errors:

$$e_D(s, a, r, s') = \sum_d |y_d - Q_d(s, a_d)|$$

where $e_D(s, a, r, s')$ denotes the error used for prioritization of the experience transition tuple (s, a, r, s').

4.4 Experiments

We evaluate the performance of the proposed BDQ agent on several challenging continuous control environments of varying action dimensionality and complexity. These environments are simulated using the MuJoCo physics engine (Todorov et al., 2012). We first study the performance of BDQ against its standard non-branching variant, Dueling DDQN, on a set of custom reaching tasks with increasing degrees of freedom and under two different granularity discretizations. We then compare the performance of BDQ against a continuous control algorithm, Deep Deterministic Policy Gradient (DDPG), on a set of standard continuous control manipulation and locomotion tasks from OpenAI MuJoCo Gym (Brockman et al., 2016; Duan et al., 2016). We also compare BDQ against a fully independent alternative, Independent Dueling Q-Network (IDQ), in order to verify our hypothesis regarding the significance of the shared network module in coordinating the distributed policies. To make the continuous-action environments compatible with the discrete-action algorithms in our study (i.e. BDQ, Dueling DDQN, and IDQ), in both sets of experiments, we discretize each action dimension $d \in \{1, ..., N\}$, in the underlying continuous action space, into n equally spaced values, yielding a discrete combinatorial action space of n^N possible actions.

4.4.1 Varying the degrees of freedom in Reacher

We begin by comparing the performance of BDQ against its standard non-branching variant, the Dueling DDQN agent, on a set of physical manipulation tasks with increasing action dimensionality. These environments, shown in Figure 4.3, are custom variants of the standard Reacher-v1 task from OpenAI Gym, which features more actuated joints ($N = \{3, 4, 5\}$) and constraints on their ranges of motion to

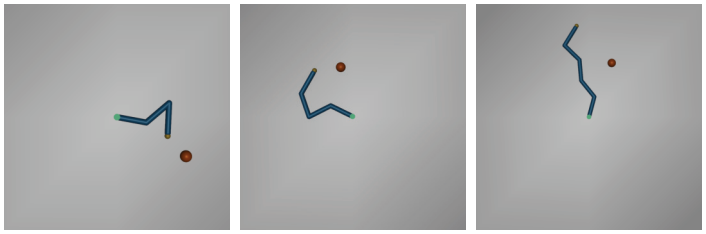


Figure 4.3: Top view of the custom Reacher environments. The green joint is fixed in space and can only rotate, between each pair of blue segments is a hinge joint, and the goal is to bring the gray end effector towards the red target. From left to right: Reacher3DOF, Reacher4DOF, and Reacher5DOF environments with 3, 4, and 5 degrees of freedom, respectively.

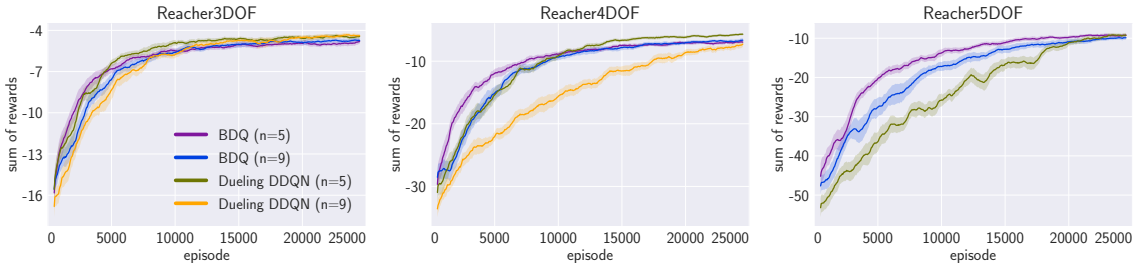


Figure 4.4: Performance in sum of rewards during evaluation on the y-axis and training episodes on the x-axis. The solid lines represent smoothed averages (window size of 20 episodes) on 3 runs with random initialization seeds, while the shaded areas show the standard deviations. Evaluations were conducted every 50 episodes of training for 30 episodes with a greedy policy.

prevent collision between segments. Unlike in the original Reacher-v1 task, reaching the target position immediately terminates an episode without the need to decelerate and maintain position at the target. These modifications to the task were selected to allow faster experimentation and simplify learning when increasing the number of actuated joints. We consider two discretization resolutions resulting in $n = 5$ and $n = 9$ sub-actions per joint. This is done in order to examine the impact of finer granularity, or equivalently more discrete sub-actions per action dimension, with increasing degrees of freedom. The general idea is to empirically study the effectiveness of action branching in the face of increasing action-space dimensionality as compared to the standard non-branching variant. Therefore, the tasks are designed to have sufficiently small action spaces for a standard non-branching algorithm to still be tractable for the purpose of our evaluations.

The performances are summarized in Figure 4.4. The results show that in the low-dimensional reaching task with $N = 3$, all agents learn at about the same rate, with slightly steeper learning curves towards the end for Dueling DDQN. In the task with $N = 4$, we see that the Dueling DDQN agent with $n = 5$ starts off less efficiently (i.e.

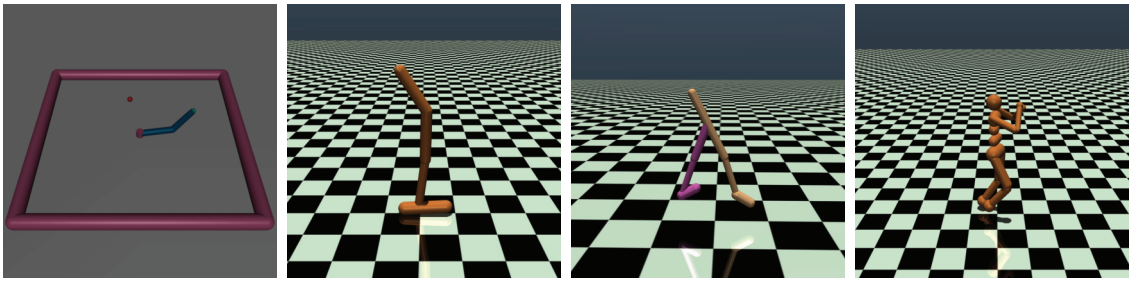


Figure 4.5: Illustrations of the OpenAI Gym tasks that were used in our experiments. From left to right: Reacher-v1, Hopper-v1, Walker2d-v1, and Humanoid-v1 featuring 2, 3, 6, and 17 degrees of freedom, respectively.

Environment	$\dim(\boldsymbol{o})$	N	n^N	$n \times N$
Reacher-v1	11	2	1.1×10^3	66
Hopper-v1	11	3	3.6×10^4	99
Walker2d-v1	17	6	1.3×10^9	198
Humanoid-v1	376	17	6.5×10^{25}	561

Table 4.1: Dimensionality of the considered environments: $\dim(\boldsymbol{o})$ denotes the observation dimensions, N is the number of action dimensions, and n^N indicates the number of possible actions in the combinatorial action space, with $n = 33$ sub-actions per action dimension. The rightmost column indicates the total number of network outputs required for the proposed action branching.

slower learning curve) than its corresponding BDQ agent, but eventually converges and outperforms both BDQ agents in their final performance. However, in the same task, the Dueling DDQN agent with $n = 9$ shows a significantly less efficient learning performance compared to its BDQ counterpart. In the high-dimensional reaching task with $N = 5$, we see that the Dueling DDQN agent with $n = 5$ performs rather poorly in terms of its sample efficiency. For this task, we were unable to run the Dueling DDQN agent with $n = 9$ since running it was computationally expensive due to the large number of actions that need to be explicitly represented by its network (i.e. $9^5 \approx 6 \times 10^4$) and consequently the extremely large number of network parameters that need to be trained at every iteration. In contrast, in the same task, we see that BDQ performs well and converges to good policies with robustness against the discretization granularity.

4.4.2 Continuous control benchmark

Here, we evaluate the performance of BDQ on a set of standard continuous control benchmark tasks from OpenAI Gym. Figure 4.5 demonstrates sample illustrations of the environments used in our experiments. Table 4.1 states the dimensionality

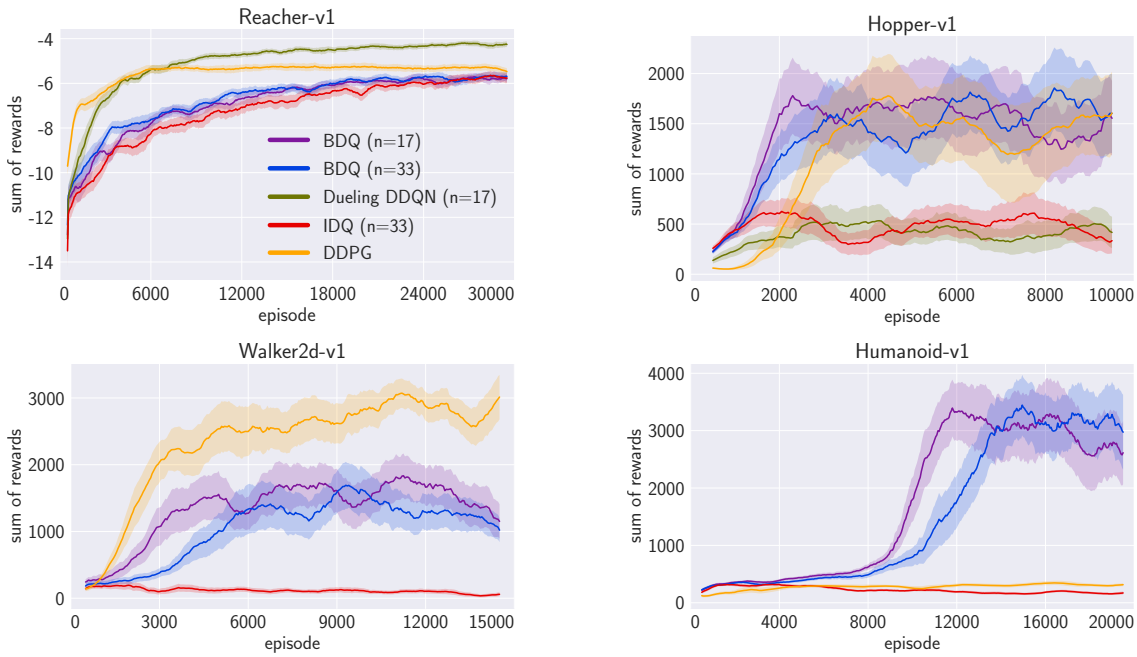


Figure 4.6: Learning curves for OpenAI Gym manipulation and locomotion benchmark tasks. The solid lines represent smoothed averages (window size of 20 episodes) over 6 runs with random initialization seeds, while the shaded areas show the standard deviations. Evaluations were conducted every 50 episodes of training for 30 episodes with a greedy policy.

information of these tasks, provided for the specific case of $n = 33$ being the finest granularity we experimented with.

We compare the performance of BDQ against a continuous-action reinforcement learning algorithm, DDPG, as well as against a completely independent agent, IDQ. For all environments, we evaluate the performance of BDQ with two different discretization resolutions resulting in $n = 17$ and $n = 33$ sub-actions per degree of freedom. We do this to compare the relative performance of BDQ for the same environments with substantially larger discrete action spaces. Where feasible (i.e. Reacher-v1 and Hopper-v1), we also run the Dueling DDQN agent with $n = 17$.

The results demonstrated in Figure 4.6 show that IDQ's performance quickly deteriorates with increasing action dimensionality, while BDQ continues to perform competitively against DDPG. Interestingly, BDQ significantly outperforms DDPG[1] in the most challenging environment, the Humanoid-v1 task which involves 17 action dimensions, leading to a combinatorial action space of approximately 6.5×10^{25} possible actions for $n = 33$. Our ablation study on BDQ (with a shared network

[1]It is worth noting that the results presented here were obtained in 2017 using the rllab library. Newer versions of DDPG and Humanoid can now produce better results, as shown in Subsection 12.4.1.

module) and IDQ (no shared network module) verifies the significance of the shared module in coordinating the distributed policies, and thus enabling the BDQ agent to progress in learning and to converge to good policies in a stable manner. Furthermore, remarkably, to perform competitively against a continuous control algorithm in such high-dimensional environment is a feat previously considered intractable for discrete-action algorithms (Lillicrap et al., 2015; Schulman et al., 2017). However, in the simpler tasks DDPG performs better or on par with BDQ. We think a potential explanation for this could be the use of the Ornstein-Uhlenbeck (Uhlenbeck & Ornstein, 1930) action noise by DDPG which creates temporally correlated exploration.

By comparing the performance of BDQ for $n = 17$ and $n = 33$, we see that, despite the significant difference in the total number of possible actions, the proposed agent continues to learn rather efficiently and converges to similar final performance levels. An interesting point to note is the exceptional performance of Dueling DDQN for $n = 17$ in Reacher-v1. Yet, increasing the action dimensionality by only one degree of freedom (from $N = 2$ in Reacher-v1 to $N = 3$ in Hopper-v1) renders the Dueling DDQN agent ineffective. Finally, it is noteworthy that BDQ is highly robust against the specifications of the TD target and loss function, while it highly deteriorates with the ablation of the prioritized replay. Characterizing the role of the prioritized experience replay, in stabilizing the learning process for action branching networks, remains the subject of future research.

4.4.3 Experimental details

Here, we provide information on the technical details and hyperparameters used to train the agents in our experiments. Common to all agents, training always started after the first 10^3 steps and, thereafter, we ran one step of training at every time step. We did not perform tuning of the reward scaling parameter for either of the algorithms and, instead, used each environment's raw rewards. We used the OpenAI Baselines (Dhariwal et al., 2017) implementation of DQN as the basis for the development of all the DQN-based agents.

BDQ We used the Adam optimizer (Kingma & Ba, 2014) with a learning rate of 10^{-4}, $\beta_1 = 0.9$, and $\beta_2 = 0.999$. We trained with a batch size of 64 and a discount factor $\gamma = 0.99$. The target network was updated every 10^3 time steps. We used the rectified non-linearity (or ReLU) (Glorot et al., 2011) for all hidden layers and linear activation on the output layers. The network had two hidden layers with 512 and 256 units in the shared network module and one hidden layer per branch with

128 units. The weights were initialized using the Xavier initialization (Glorot & Bengio, 2010) and the biases were initialized to zero. A gradient clipping of size 10 was applied. We used the prioritized replay with a buffer size of 10^6, $\alpha = 0.6$, and linear annealing of β from $\beta_0 = 0.4$ to 1 over 2×10^6 steps.

While an ϵ-greedy policy is often used with Q-learning, random exploration (with an exploration probability) in physical, continuous-action environments can be inefficient. To explore well in physical environments with momentum, such as those in our experiments, DDPG uses an Ornstein-Uhlenbeck process (Uhlenbeck & Ornstein, 1930), which creates a temporally correlated exploration noise centered around the output of its deterministic policy. The application of such a noise process to discrete-action algorithms is, nevertheless, somewhat nontrivial. For BDQ, we decided to sample actions from a Gaussian distribution with its mean at the greedy actions and with a small fixed standard deviation throughout the training to encourage lifelong exploration. We used a fixed standard deviation of 0.2 during training and zero during evaluation. This exploration strategy yielded a mildly better performance as compared to using an ϵ-greedy policy with a fixed or linearly annealed exploration probability. However, for the custom reaching tasks, we used an ϵ-greedy policy with a linearly annealed exploration probability, similar to that commonly used for Dueling DDQN.

Dueling DDQN We generally used the same hyperparameters as for BDQ. The gradients from the dueling streams were rescaled by $1/\sqrt{2}$ before entering the shared feature module as recommended by Wang et al. (2016). The average aggregation method was used to combine the state value and advantages as originally reported for the best-performing agent. We experimented with both a Gaussian and an ϵ-greedy exploration policy with a linearly annealed probability, and observed a moderately better performance for the later, used in the reported results.

IDQ Once more, we generally used the same hyperparameters as for BDQ. Similarly, the same number of hidden layers and hidden units per layer were used for each independent network, with the difference being that the first two hidden layers were not shared among the several networks (which was the case for BDQ). The dueling architecture was applied to each network independently (i.e. each network had its own state-value estimator). This agent serves as a baseline for investigating the significance of the shared module in the proposed action branching architecture.

DDPG We used the implementation of DDPG from the rllab library (Duan et al., 2016) and the hyperparameters reported by Lillicrap et al. (2015), with the exception

of not including an L_2 weight decay for Q, as opposed to the originally proposed penalty of 10^{-2}, which deteriorated the performance in our experiments.

4.5 Discussion

This work has demonstrated how agent components can be designed to address a specific reinforcement learning problem. In particular, we focused on adapting Q-learning techniques to large multidimensional discrete action spaces. We introduced a new neural network architecture that distributes a value function representation across multiple branches, while maintaining a shared encoder to allow for some form of implicit centralized coordination. We have adapted the DQN algorithm, as well as several of its more notable extensions, into the proposed action branching architecture and demonstrated its effectiveness. We believe that the proposed approach can be applied in a wide range of situations involving large multidimensional action spaces. However, further work is needed to better appreciate its limitations and, in particular, to understand the game-theoretic aspects of semi-independent updates.

Chapter 5

Learning to reach spatial goals

This chapter is based on the paper "Scaling all-goals updates in reinforcement learning using convolutional neural networks" (Pardo et al., 2020) published at the AAAI 2020 conference. A preliminary version was presented under the name "Goal-oriented trajectories for efficient exploration" at the ICML 2018 exploration in RL workshop (ERL) and a second version was presented under the name "Q-map: A convolutional approach for goal-oriented reinforcement learning" at the NeurIPS 2018 deep RL workshop.

The videos are available at: https://sites.google.com/view/q-map-rl.
The source code is available at: https://github.com/fabiopardo/qmap.

The main research questions are:

- How to efficiently learn to reach spatial goals?
- Can we achieve an effective exploration method by reaching goals?

5.1 Introduction

Reinforcement learning environments can typically be classified as goal-reaching and reward-based types. In the former case, tasks like navigating a maze directly justify the use of policies and value functions that are conditioned simultaneously on the observations of the world and on goals. In the latter case, maximizing the discounted sum of future rewards provides a more subtle objective that can describe any given task. Although learning can be more challenging in this case, it can sometimes be greatly simplified by learning task-agnostic, goal-directed skills in combination with a higher-level task-dependent module, responsible for selecting intermediate goals to reach (Bakker & Schmidhuber, 2004; Kulkarni et al., 2016; Vezhnevets et al., 2017; Peng et al., 2017).

In both cases, learning to reach each goal independently would be very inefficient.

However, because the goals conditioning the policy do not modify the dynamics of the environment, off-policy algorithms can greatly improve the sample efficiency by using *goal relabeling*. When replaying a past transition, this technique substitutes the given goal with another valid one, in order to evaluate the action with respect to the new goal. Multiple strategies exist for selecting these goals. For example, in hindsight experience replay (HER) (Andrychowicz et al., 2017), the "future strategy" selects those goals randomly within the list of future states visited in the episode. Other strategies rely on a uniform distribution (Schaul et al., 2015a; Veeriah et al., 2018) or generative models (Nair et al., 2018). However, in the case where all the possible goals can be enumerated, a single transition can be maximally used by updating the policy towards *all-goals*.

A traditional goal-conditioned network requires one forward pass per goal which can be intractable in the case of large goal sets. Instead, we propose to use a network producing Q-values for all combinations of actions and goals, at once. Given a single observation, a stack of "Q-frames" are produced in output, one for each action, representing a measure of the number of steps required to reach each goal cell. To allow the model to exploit spatial correlations and to keep the total number of parameters small, we propose to generate those frames using convolutional layers.

We evaluate the accuracy and generalization properties of the proposed network on tasks consisting in finding the shortest path towards all points in random mazes and solving Sokoban puzzles. Our results show that the proposed approach converges faster and achieves better final performance while being more stable than the goal-in-input approach. We then demonstrate that the proposed network learns well in more visually complex and difficult to control environments. On Montezuma's Revenge we show that the capacity to query this large number of Q-values can allow an agent to explore well by selecting random but feasible goals. Finally, on Super Mario All-Stars, we show that when coupled with a task-learner DQN agent (Mnih et al., 2015) the exploration method significantly improves the performance.

5.2 Related work

5.2.1 Goal-oriented value functions

Generalized value functions While a state-action value function is usually specific to the rewards defining the task, the generalized value functions (GVFs) (Sutton et al., 2011) $Q_g^\pi(s, a)$ are trained with pseudo-reward functions $r_g(s, a, s')$ that are specific to each goal g. The Horde architecture combines a large number of independent GVFs trained to predict the effect of actions on sensor measurements

and can be simultaneously trained off-policy.

Universal value function approximators The universal value function approximators (UVFAs) (Schaul et al., 2015a) extend the concept of GVFs by adopting a unique state-action value function $Q^\pi(s, a, g)$ parameterized by states, actions, and goals together, enabling interpolation and extrapolation between goals.

5.2.2 Goal relabeling strategies

Goal relabeling strategies rely on the fact that a single transition can be used to update the policy towards different goals. A few different methods for selecting these goals have been proposed, some of which are explained below.

Future states In hindsight experience replay, the goals are sampled from the list of subsequent states traversed in the remainder of the episode when selecting a transition. This allows to quickly propagate successful examples of goal reaching as all of the required transitions are present in the recorded episode. However, this approach limits the scope of goals considered.

Random states A simpler approach is to select goals uniformly over the set of goals which can be known or discovered through interaction (Schaul et al., 2015a; Veeriah et al., 2018). This approach can potentially be more general but it can also suffer more from instability and slow learning if a number of goals are unreachable or not useful.

Imagined goals from a generative model If the set of possible states is unknown and we wish to get more diverse goals than the ones discovered so far, it is possible to learn a generator from which the goals can be sampled. For example, a variational autoencoder (VAE) (Kingma & Welling, 2013) can be trained to learn the distribution of states and used to generate new plausible goals (Nair et al., 2018). This approach can lead to more general-purpose goal-reaching policies but relies on the accuracy of the learned generator. Therefore, a mixture of imagined and random goals can be more advantageous (Nair et al., 2018).

All goals The previous approaches are sometimes referred to as *many-goals learning* (Veeriah et al., 2018) and are general enough to work with continuous and unknown goal spaces. However, many tasks of interest have a known finite number of goals. In which case, the best use of a single transition can be achieved by updating the policy towards all goals, hence the name *all-goals updates*. For example, an

independent Q-value for each of the goals can be learned separately with tabular Q-learning (Kaelbling, 1993b). The main issue with this approach is that the number of updates scales linearly with the number of goals.

5.2.3 Using convolutions to represent value functions

Training several function approximations in parallel quickly becomes impractical in the case of representing many value functions. Multiple works showed how a shared torso fully connected to multiple heads can be used to learn several value functions (Osband et al., 2016; Van Seijen et al., 2017; Cabi et al., 2017) as is the case for example in Chapter 4. A few approaches aim at representing value functions using convolutional neural networks, some of which are described below.

RL with unsupervised auxiliary tasks In UNREAL (Jaderberg et al., 2016), the main policy is trained along multiple unsupervised auxiliary policy heads to help generate useful features. The pixel-control task trains the agent to predict which actions generate the most change in the input frames. The head specific to this task uses transposed convolutions to generate a three dimensional array of Q-values. These values were however never used to control the agent and were not specialised in goal-reaching.

Value iteration networks A value iteration network (VIN) module (Tamar et al., 2016) creates a reward frame which is fed into a convolutional layer generating Q-values which are then max-pooled along the action channel to produce a frame which is iteratively fed into the module until it represents the state-values of the optimal policy. An attention mechanism is then applied on the value frame to create features for a policy trained via reinforcement learning. A VIN module can in theory perform planning as demonstrated on goal-reaching tasks. While VIN is related to our model it has many differences. First, it does not directly use the generated values to act. Second, for goal-reaching tasks, those values are the expected returns when starting from each state location for a goal given in input while our approach represents the values of starting from a given state to reach all the possible goals. Finally, VIN performs planning by iteratively recomputing the values at every step while our approach is based on learning a direct mapping from inputs to values.

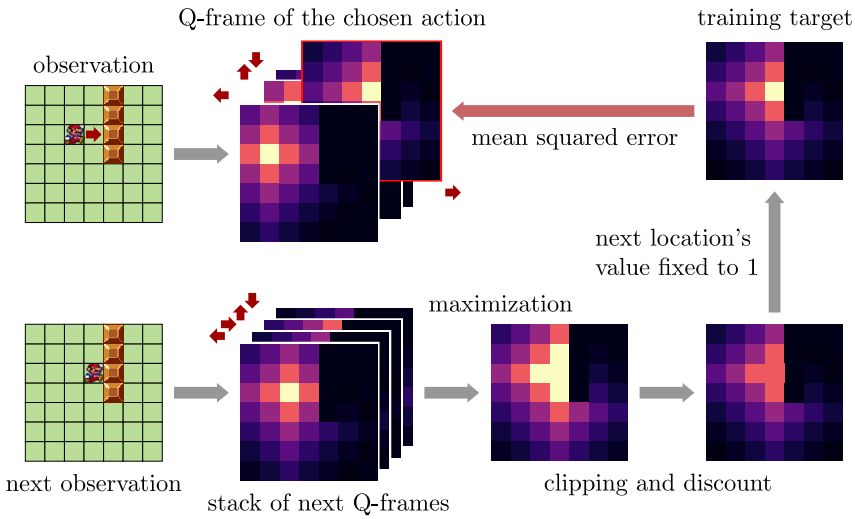


Figure 5.1: Training process for a Q-map model updating the prediction towards all goals at once. The Q-frame of the chosen action is updated with Q-learning, via the stack of Q-frames generated at the next observation.

5.3 Methods

5.3.1 Predicting the distance towards all goals with Q-map

While UVFA-like architectures take observations and goals in input and generate Q-values for every action in output, our model, named *Q-map*, only takes observations in input and generates Q-values for every goal and action in output. For example, in the case of a grid of 2D goal coordinates, when a stack of observation frames are provided in input, another stack of 2D *Q-frames* are generated where the rows and columns represent the goal locations and the number of frames represents the number of actions. Note that the height and width of the observations and generated frames can be different for example if the observations require more resolution as shown in some of the following experiments. It is also worth noting that inputs do not need to be frames, they could for example be state vectors, even if the experiments included in this work focused on pixel-based observations.

The output values represent Q-values in the context of the goal-reaching reward function awarding 1 with episode-termination at the goal and 0 otherwise. The discount factor γ creates exponentially decaying values, γ^{k-1} indicating the number of steps k to the goal. A value of 0 indicates that the goal cannot be reached, for example when a goal is in an obstacle, and 1 means that the goal will be reached at the next time step.

The training of a Q-map, illustrated in Figure 5.1, relies on a multidimensional version of Q-learning. For a given transition, only the output frame corresponding

to the taken action is updated using a mean-squared error with a target frame. This target is generated as follows: First, a forward pass in the model is performed with the next observation in input. Then, the generated frames are maximized over the action dimension to generate a single frame representing a measure of the minimum expected number of steps towards each of the goals. This frame is then clipped to the range $(0, 1)$ to reduce the under- and over-estimations. Finally, the frame is discounted by γ and the location reached at the next step is set to 1. This procedure effectively uses one environmental transition to virtually create a set of independent one-step episodes, one per goal, with termination on success and partial-episode bootstrapping otherwise as described in Chapter 10.

To use the generated Q-frames to reach a specific goal, one only needs to take the vector of Q-values at the goal location and select the action of maximum value.

5.3.2 Neural network architectures

We considered multiple neural network architectures to support the proposed model. The first choice, which will be our baseline in the rest of this work, is a UVFA-like network which takes a stack of observation and a goal frame and generates the corresponding vector of Q-values, one for each action. To create a many-goals or all-goals update, the batch needs to consists of frames where the observation remains the same but the goal, represented by a one-hot frame, changes to cover all desired values. The batch output is then a set of vectors which can be reshaped to create the expected stack of Q-frames.

The second network uses an autoencoder-inspired architecture, with convolutions followed by fully-connected layers, in turn followed by transposed convolutions. This network takes observation frames in input and generates the full Q-frames in output. The convolutional nature of this approach benefits from a potentially better capacity to share features and use correlations between the vision of the environment and the expected number of steps.

Finally, the last network architecture that we considered is composed solely of convolutions without compression and decompression. For example if the height and width of the Q-frames corresponds to the ones of the observations, strides 1 and padding "same" are used to keep the shape of the feature maps constant, not loosing any localization information.

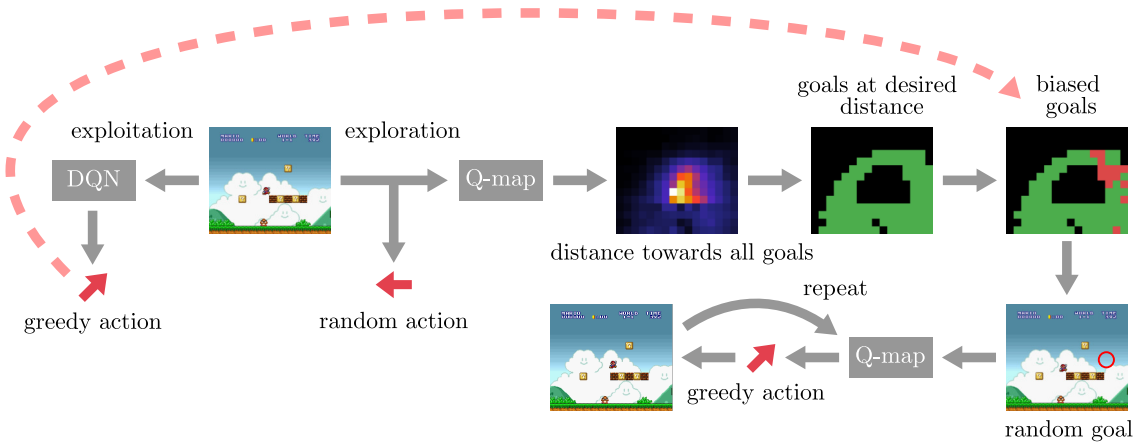


Figure 5.2: Illustration of the action-decision pipeline of a DQN agent with the proposed exploration method.

5.3.3 Exploring by reaching random goals

As a simple application case for a Q-map, we propose an exploration method, illustrated in Figure 5.2, that replaces the noisy random actions frequently used to explore in reinforcement learning, with a sequence of steps towards a goal. A proposed agent for example uses Q-map to explore and DQN to exploit. At each time step, the action selection follows a decision tree: with probability ε_r a completely random action can be selected to ensure pure exploration for Q-map and DQN. If no random action has been selected and no goal is currently chosen, a new one is chosen with probability ε_g by first querying Q-map for the predicted distance towards all the goals, filtering the ones too close or too far away (based on two hyper parameters specifying the minimum and maximum number of steps), choosing one at random, and setting a time limit to reach it based on the predicted distance. If a goal is currently chosen, the Q-map is queried to take a greedy action in the direction of the goal and the time allotted to reach it is reduced. Finally, if no random action or goal has been chosen, DQN is queried to take a greedy action.

Because the number of steps spent following goals is not accurately predictable, ensuring that the average proportion of exploratory steps (random and goal-oriented actions) follows a scheduled proportion ε is not straightforward. To achieve a good approximation, we dynamically adjust the probability of selecting new goals ε_g by using a running average of the proportion of exploratory steps $\tilde{\varepsilon}$ and increasing or decreasing ε_g to ensure that $\tilde{\varepsilon}$ approximately matches ε. This allows us later to compare the performance of our proposed agent and a baseline DQN with a similar proportion of exploratory actions.

Furthermore, unlike with the ε-greedy approach, where an action is either entirely

random or greedy, it is possible to bias the exploratory actions towards greedy actions by selecting a goal such that the first action towards the goal is the same as the greedy action proposed by the task-learner agent. Such bias aims to reduce the "cancelling-out" actions usually experienced with random exploration, while still widening the exploration horizon towards the task-learner's intended direction.

Finally, while we do not expect this exploration method to outperform specialised ones that use a variety of intrinsic signals, such as information gain (Kearns & Koller, 1999; Brafman & Tennenholtz, 2002), state visitation counts (Bellemare et al., 2016; Tang et al., 2017) or prediction error (Stadie et al., 2015; Pathak et al., 2017), it can be incorporated into most of those or used as a drop-in replacement for ε-greedy. It is worth noting that other works proposed to base the exploration on goal selection (Baranes & Oudeyer, 2013; Florensa et al., 2017b; Péré et al., 2018; Colas et al., 2018) but to the best of our knowledge none of them relied on generating large and consistent steps in the environment using an auxiliary goal-reaching policy.

5.4 Experiments

In Subsection 5.4.1 and Subsection 5.4.2 we aim to evaluate the accuracy, training time and generalization properties of the proposed Q-map model on gridworld environments, while in Subsection 5.4.3 and Subsection 5.4.4 we test Q-map in more visually complex environments and propose an application to exploration in reinforcement learning. In all of the experiments we use $\gamma = 0.9$ for the goal-reaching Q-functions and the neural networks are described using the notations: conv(filters, kernel sizes, strides) for convolutions (with padding "same" unless stated otherwise), deconv2d for transposed convolutions and dense(units) for dense, fully connected layers. Elu activation functions are used for every layer except for the output ones.

5.4.1 Evaluating the accuracy of Q-map on random mazes

The environment consists of a single pixel that can be moved in cardinal directions in a 16×16 area with traversable pathways surrounded by walls and generated such that any two points in the maze are only connected by one path. Actions towards walls result in the pixel remaining stationary. The observations consist of a stack of 16×16 RGB frames of the full view of the maze. The background is white, walls are black while the controlled pixel is in red. For the baseline model, we use a green pixel to represent the goal.

We consider three dueling double DQN architectures for evaluation (Wang et al., 2016; Van Hasselt, 2010; Mnih et al., 2015). The baseline architecture "goal-in-input"

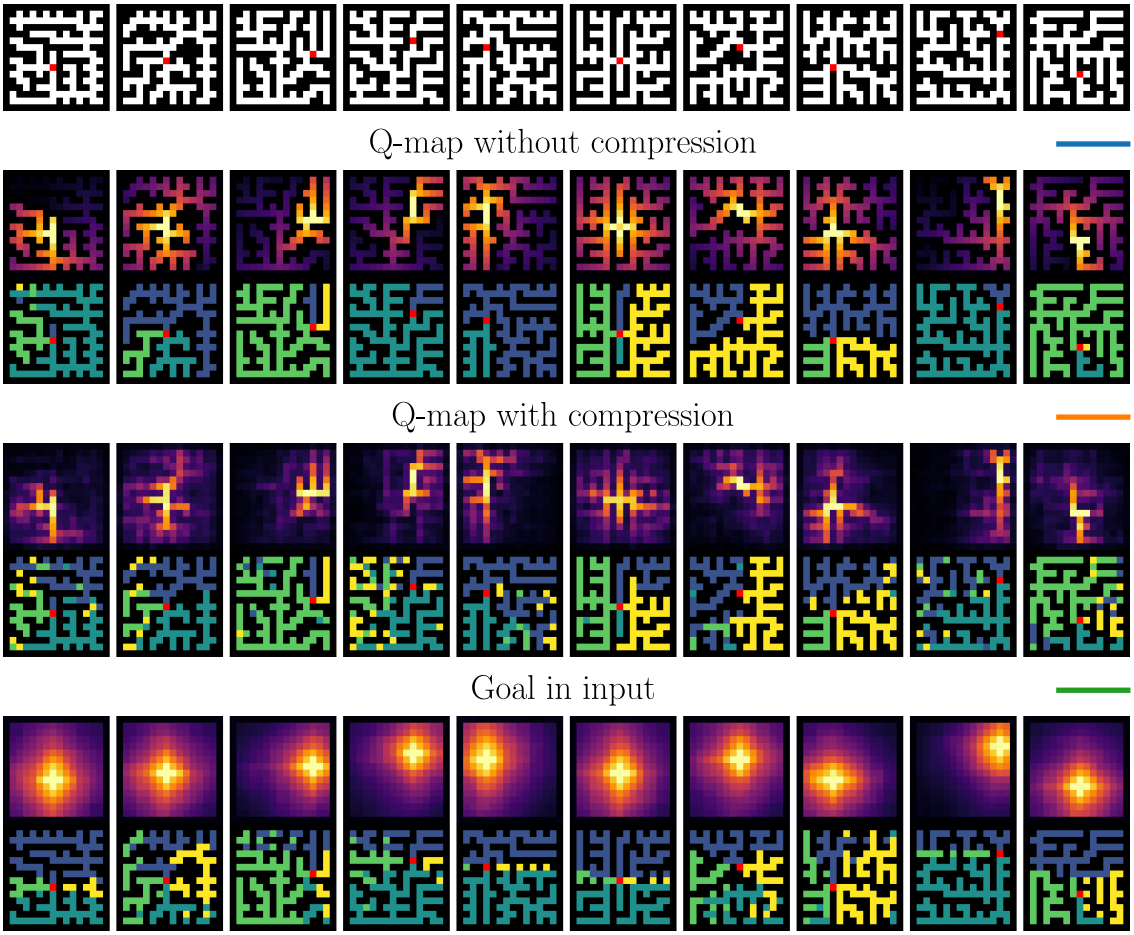


Figure 5.3: Visualization of max Q-frames obtained on random mazes using different methods. The first row shows a set of 10 random mazes with one starting point, shown in red. Then, for each agent, the first row shows the maximum Q-frames (maximum over the action dimension) generated, and the second row shows the greedy action (argmax over the action dimension) towards every goal where green means left, turquoise is down, yellow is right and blue is up.

uses 2×conv(64, 4, 2)-dense(512) layers followed by a dense(256)-dense(4) advantage branch and a dense(256)-dense(1) state-value branch. The online network (not counting the target one) uses approximately 860K parameters. The second architecture Q-map "with compression" uses 2×conv(64, 4, 2)-dense(512)-dense(1024) layers followed by a deconv(64, 4, 2)-deconv(4, 4, 2) advantage branch and a deconv(64, 4, 2)-deconv(1, 4, 2) state-value branch. The online network uses approximately $1,260$K parameters. The third architecture Q-map "without compression" is composed of 4×conv(64, 4, 1) layers followed by a 3×deconv(64, 4, 1)-deconv(4, 4, 1) advantage branch and 3×deconv(64, 4, 1)-deconv(1, 4, 1) state-value branch. The online network uses approximately 800K parameters.

Instead of letting the models interact with the environment, we choose to generate

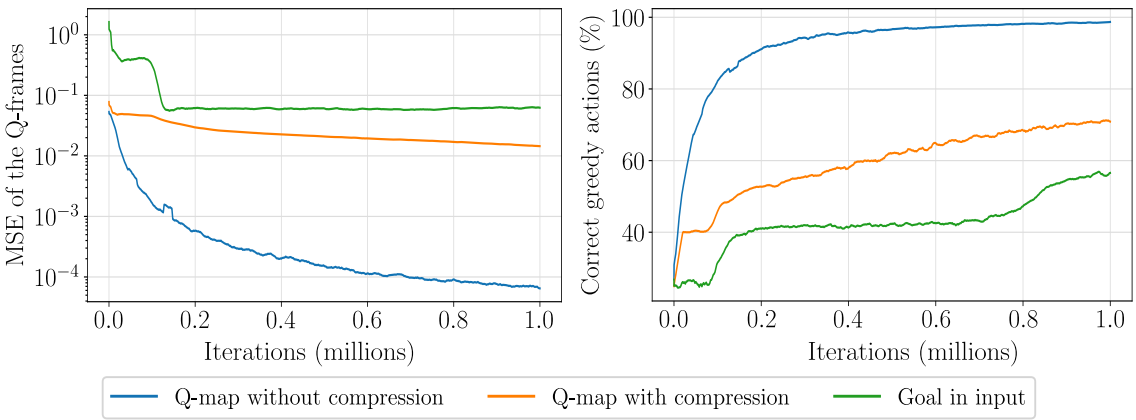


Figure 5.4: Performance of Q-maps obtained on random mazes using different architectures. Left: Mean squared error between the generated maximum Q-frames and the ground truth generated by path-finding. Right: Percentage of correct greedy actions to take towards each possible goal. The Q-map architecture performs better than the baseline in both measures and the Q-map with no compression shows very fast learning and generalization qualities.

a training and testing sets in order to remove any exploration bias when comparing the quality of the Q-frames generated. The training set is comprised of all possible transitions in $2,317$ different mazes for a total of $1,000,220$ transitions. The testing set is comprised of all possible starting points in 10 new mazes for a total of $1,075$ observations. We use batches of 50 random transitions from the training set. The loss is the mean squared error between the produced Q-values and the target ones. The evaluation metric is the "success rate" which is a proportion of state-goal pairs where the models correctly predict the greedy action towards all feasible goals, obtained by using the $\arg\max$ operator over the action dimension.

The Figure 5.3 shows examples of the learned Q-frames for the test mazes while Figure 5.4 provides the error and success rate curves for Q-map without and with compression and the baseline using goals in input. The Q-map "without compression" architecture clearly outperforms the alternative, achieving 99% success rate at $1M$ training iterations. The "Goal in input" architecture has learned what one could consider a naive first-pass solution to the task, that is a fading gradient centered on the agent's location in the maze, but failed to recognize walls of the maze. The Q-map "with compression" based architecture performed significantly better but still struggled to produce sufficiently sharp Q-frames, unable to determine correct action choices outside of close proximity to the agent. It is worth noting that the Q-values for the maze walls are not specifically masked and the Q-map successfully learns to decay them to 0. Videos showing the Q-frames and actions through the training of all three architectures are available on the website.

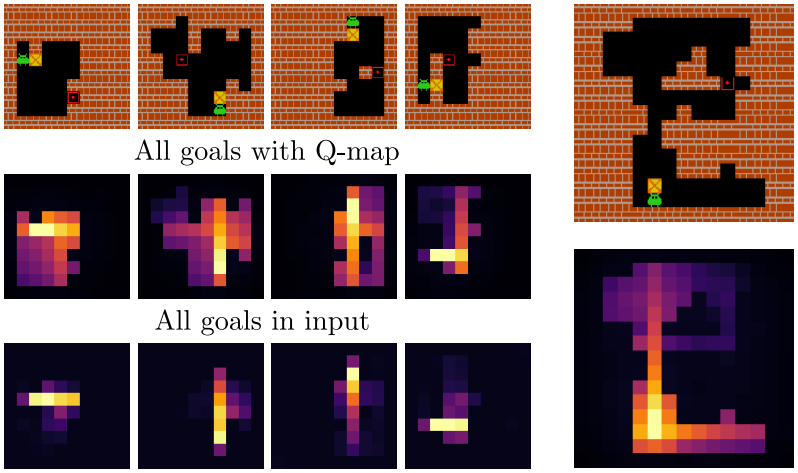


Figure 5.5: Visualization of max Q-frames obtained on Sokoban using different methods. Left: Max Q-frames learned by Q-map and the all-goals in input approaches on unseen 10×10 puzzles. Right: A separate example from a Q-map trained on 15×15 levels.

5.4.2 Comparing update strategies on Sokoban

In this section, we consider a notoriously difficult environment suited for planning (Racanière et al., 2017): Sokoban, from Gym-Sokoban (Schrader, 2018). A significant difference to the previously considered Maze environment is that the goal is specified for a box which has to be carefully pushed by an agent-controlled avatar.

This time, we compare the Q-map "without compression" model with two goal-in-input baselines. The two baselines differ by their goal-relabeling strategies: one uses random goals while the other one uses all goals. All the models are trained on the same batch size of 100. For the all-goals updates model, because there are 100 possible goals, a batch contains a single observation replicated 100 times with each of the possible goals.

Both of the baseline models use a 5×conv(64, 5, 1)-2×conv(64, 5, 1, valid)-dense(256)-dense(4) network, while the Q-map uses a 7×conv(64, 5, 1)-conv(4, 5, 1) network. We keep the total number of parameters roughly the same between the models with approximately 688K for the baselines and 626K for the Q-map.

The models are provided with 10×10 RGB frames representing the walls, box and the agent's locations in red, green and blue respectively. The training set, identical between the models, is generated by performing exclusively random actions. During evaluation, the greedy actions are followed and the success rate measures the proportion of solved puzzles.

Multiple generated Q-frames examples are given in Figure 5.5. The Q-map model

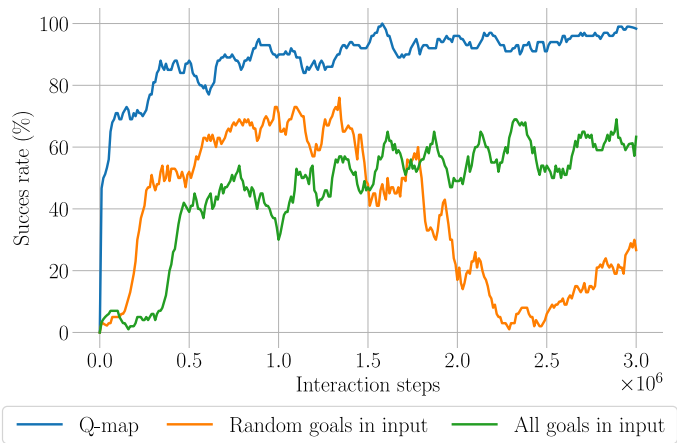


Figure 5.6: Success rate on unseen Sokoban puzzles. Q-map performs better than the goal-in-input baselines. The random-goals baseline is particularly unstable.

was able to learn more reliable Q-values, showing how the box should be moved even in complex scenarios requiring a sequence of actions with long-term dependencies. Figure 5.6 shows the success rate of each of the considered models. Each point on the graph represents an average of 10 individual evaluation runs. The Q-map model straight away outperforms both of the baselines and keeps improving until almost perfect performance in a very stable way. Of the two baselines, the one trained with random-goals initially seems to learn faster but then deteriorates, suggesting that learning with all goals simultaneously is more stable.

Finally, we note that the Q-map model could in theory support multiple boxes by increasing the dimensionality of the outputs to account for the combinatorial nature of the problem.

5.4.3 Exploring with Q-map on Montezuma's Revenge

So far we have demonstrated the efficiency of the proposed Q-map model in a selection of gridworld problems which, while challenging to solve, have very simple state spaces and action dynamics. In this section we use the Montezuma's Revenge (Atari 2600) game to evaluate the learning capability of Q-map and roughly compare the exploration boundaries between a random-action policy and a random-goal policy that utilizes Q-map for action selection. Environmental rewards are ignored and no task-learner agent is present in this experiment. The Q-map is trained with the generated transitions.

The actions are limited to "no-op, left, right, jump, jump-left, jump-right, down" and are repeated four times. The game frames are zero-padded to reach a resolution of 160×224 before being scaled with a factor of $1/4$. Three grey-scale 40×56 frames,

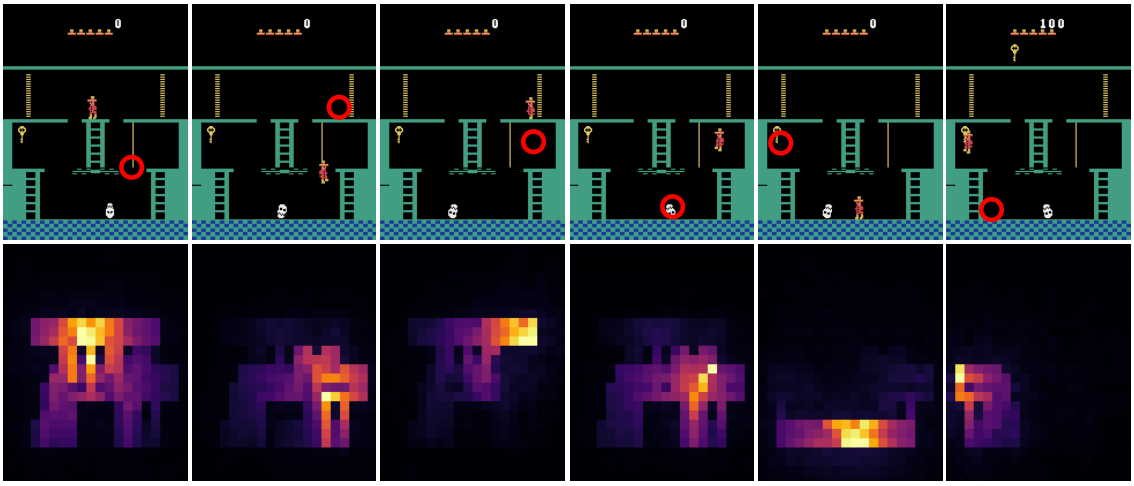


Figure 5.7: A random sequence of goal-reaching steps successfully exploring most of the first room in the Montezuma's Revenge game. The goals are shown with circles and the corresponding learned max Q-frames accurately represent reachable places.

spaced by two steps are stacked to produce the observations while another scaling factor of 1/2 is used for the coordinates, limiting the Q-frames to 20×28.

We use a Q-map "with compression" to accommodate the resolution discrepancy between the observations and the output. The network uses conv(32,8,2)-conv(32,6,2)-conv(64,4,2)-dense(1024)-dense(1024) layers followed by a deconv(32, 4, 2)-deconv(32, 6, 2)-deconv(4, 8, 1) advantage branch and a deconv(32, 4, 2)-deconv(32, 6, 2)-deconv(1, 8, 1) state-value branch. A random goal is chosen within 15 to 30 predicted steps from the agent's current position. An individual goal-directed trajectory terminates upon either reaching the goal or exceeding 150% of the original predicted number of steps. Furthermore, there is a chance to take a random action decayed linearly from 0.1 to 0.05.

Qualitatively, the learned Q-frames are sufficiently accurate for the random-goal policy to be able to navigate much further through the environment and even actively avoid the contact with the skull on the way towards a goal. Example Q-frames are shown in Figure 5.7. We also tracked the number of keys picked up by both of the exploration strategies. The random policy only reached the key once in the 5 million steps, while the random-goal policy first reached the key at the 1.2 million steps and overall 398 times in 5 million steps.

5.4.4 Combining Q-map and DQN on Super Mario All-Stars

Finally, the proposed exploration method is used as a drop-in replacement for the ε-greedy exploration in a DQN agent on the Super Mario All-Stars game (SNES)

Figure 5.8: Example of learned max Q-frames on Super Mario All-Stars. The Q-map model successfully takes into account the obstacles. The horizontal patterns are due to the action repeat skipping some locations.

(OpenAI, 2018). The actions are limited to "no-op, left, right, up-left, up, up-right" and are repeated four times. The rewards from the game are divided by 100 with 0 bonus for moving to the right or penalty for game overs. Terminations by touching enemies or falling into pits and the coordinates of Mario and of the scrolling window are extracted from the RAM and used by the environment. Episodes are naturally limited by the timer of 400 seconds present in the game, which corresponds to $2,402$ steps. Observations consist in three 56×64 grayscale frames spaced by two steps and the goal space is scaled down to 32×28.

We used the network described in Subsection 5.4.3 for Q-map. The network used for DQN has conv(32,8,2)-conv(32,6,2)-conv(64,4,2) layers followed by a dense(1024)-dense(6) advantage branch and a dense(1024)-dense(1) state-value branch. Furthermore, to account for the movement of the screen in the game, the target Q-frames are shifted accordingly before performing the updates. Example Q-frames are shown in Figure 5.8.

Similarly to Subsection 5.4.3, we initially compare a random-goal Q-map based policy with a random-action policy. The states visited during 2 million steps of interaction are displayed in Figure 5.9. As can be seen, the proposed exploration method covers a significantly larger area, allowing the agent to almost reach the end of the level without using any environmental or intrinsic reward.

We then evaluate an augmented agent that is comprised of a Q-map-based exploration and a DQN (Mnih et al., 2015) task-learner agent. For both this agent and the baseline, we use an exploratory schedule that linearly decreases from 100% to 5%. For the proposed agent, the random action probability is decreased from 10% to 5% over the course of the training. In order for the total proportion of

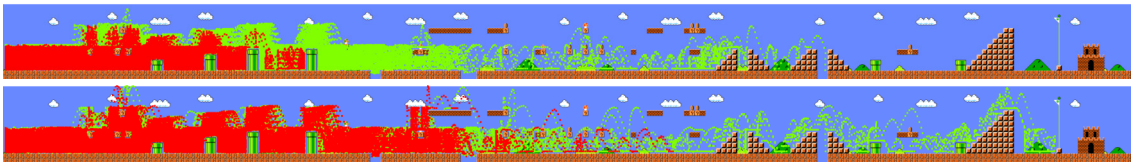


Figure 5.9: Coordinates visited after 2 million steps on the first level of Super Mario All-Stars. Top: Random walk (red) compared against the proposed Q-map random-goal walk (green). Bottom: DQN using ε-greedy (red) compared against DQN with the proposed exploration (green). In both cases, Q-map allows to explore significantly further

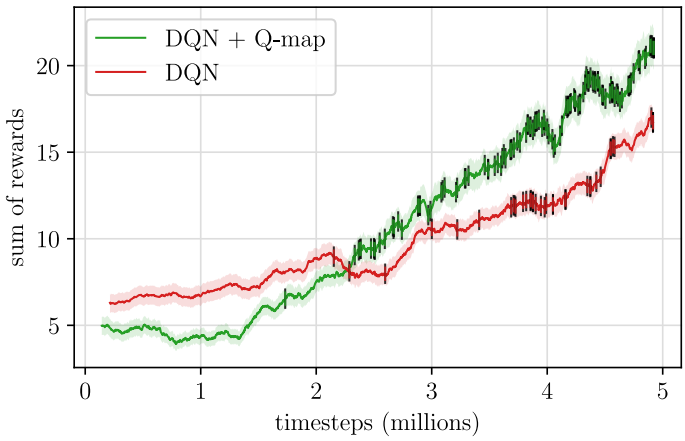


Figure 5.10: Performance comparison on Super Mario All-Stars, between ε-greedy exploration (red), and the proposed exploration (green) with confidence intervals of 99%. The vertical bars indicate flags reached. The proposed agent significantly outperforms the baseline, reaching the flag earlier and more frequently.

exploration steps to match the planned schedule, the chance to start a new goal-reaching trajectory is then dynamically adjusted. This implies that, at the end of the training, when the proportion of exploratory actions has to match the 5% chance of random action, the probability to use random goals becomes 0. Furthermore, to bias the exploration towards useful directions, a 50% chance to select the goals with a first action identical to the one from DQN is used.

To measure the performance of the proposed combined agent in terms of the sum of rewards collected per episode and the number of flags reached, we ran both agents for 5 million steps with the same four seeds (0, 1, 2, 3) and reported the results in Figure 5.10. The initial performance of the augmented agent is worse than the baseline, likely due to the fact that early rewards can easily be collected with random movements. In turn, the longer exploration horizon of the augmented agent enables it to learn to progress through the level faster and learn to reach the final flag consistently. In total, the baseline reached the flag 37 times while the proposed agent reached it 134 times with a final performance 30% higher. Finally, we also

tested the capacity of the Q-map model to adapt to a different level and found that the learning was significantly faster when transferring a pre-trained Q-map model from one level to another than when training from scratch as visible in the videos.

5.5 Discussion

We proposed an all-goals Q-learning model that is computationally efficient and able to generalise to previously unseen goals. A single environmental transition is used to generate a value estimate for the full set of possible goals in a single forward pass in the network. This is achieved by utilising a convolutional architecture with the intuition that such a model would take advantage of the correlations between the input features and the similarities of the neighbouring goals in the environment. We have demonstrated that this approach significantly outperforms goal-in-input baselines and achieves nearly perfect success rate in gridworld maze pathfinding and Sokoban tasks. In addition, we have shown that the Q-map model is capable to learn to navigate in visually complex environments such as Montezuma's Revenge and Super Mario All-Stars games. Finally, we provided an example of an application where by replacing random ε-greedy exploration actions with random goal-directed trajectories we improve the performance of a DQN agent.

In the experiments using Montezuma's Revenge and Super Mario All-Stars, we assumed that the location of the avatar was available to create the updates. While this can appear as a strong assumption, it is not unreasonable to imagine that an agent could learn to localize its avatar or other controllable objects (Moniz et al., 2019; Sawada, 2018) and use this knowledge when training a Q-map model.

Furthermore, it is worth noting that the proposed approach could be extended beyond images in observations, such as angles and velocities of objects, or point clouds, and the Q-frames could be replaced by Q-tensors to represent larger coordinate spaces architecturally achieved by using multi-dimensional convolutions. This could, for example, enable an agent to control robotic arms or flying drones.

Finally, learning to reach coordinates with Q-map can be useful in many scenarios. For example in hierarchical reinforcement learning, a high-level agent could be rewarded to provide useful goals to a low-level Q-map model. Also, a count-based exploration method could be combined with the estimated distance from a Q-map to select close and less visited coordinates. Furthermore, learning Q-map in itself could be a signal for an auxiliary task or a curiosity-based exploration strategy.

Part II

Designing environments

Chapter 6

Environment structure

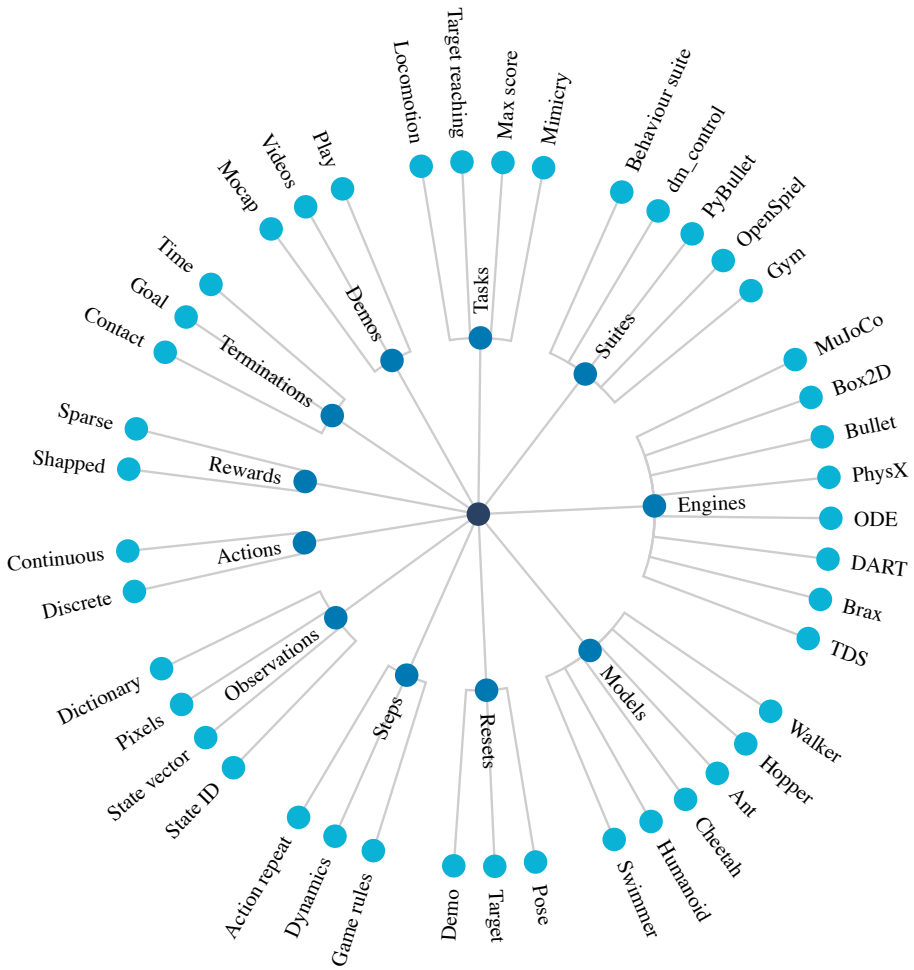


Figure 6.1: Radial layout enumerating a number of elements involved in the design of environments. These include tasks, suites, physics engines, simulation models, resets, steps, observations, actions, rewards, terminations, and demonstrations.

6.1 The environment

The environment is the agent's world. It is an algorithm of sorts that receives the agent's actions and produces the following inputs for the agent, typically observations, rewards, and terminations. Various components must be considered to build environments. Some of the main groups are shown in Figure 6.1 and are described in the next sections. Although the terms "environments", "domains", and "tasks" are sometimes used interchangeably, this thesis uses the following nomenclature.

Tasks Tasks are the objective that agents must optimize. Examples include moving as quickly as possible in a given direction, reaching a target, or imitating certain demonstrations. To convey this goal in a way that can be understood by an agent, rewards, terminations, and observations can be used.

Domains Domains are the context, such as which body to control and in which scenario. The term *domain adaptation* describes the ability to handle changes, such as object color, camera angle, or even changes in dynamics, as is the case when transferring from simulation to reality.

Environments Environments are the combination of domains and tasks, they are the whole algorithm that the agent interacts with.

Suites Suites are sets of environments with a generally unified interface. The Arcade Learning Environment (ALE) (Bellemare et al., 2013) suite was created for researchers and hobbyists to develop artificial intelligence systems for Atari 2600 games. It was built on the Atari 2600 Stella emulator. The DQN paper (Mnih et al., 2015) contained an important benchmark using ALE and paved the way for evaluating deep reinforcement learning agents against other agents and humans. OpenAI Gym (Brockman et al., 2016) is arguably the most widely used suite in the deep reinforcement learning field, wrapping ALE games and containing a number of other environments, including continuous control and toy problems.

6.2 Environment components

Physics engines Most continuous control environments require complex physics engines to accurately simulate the laws of physics, describing the movement and interaction of objects. MuJoCo (Todorov et al., 2012) is a good example of a physics engine. It is used by the Gym and dm_control suites for its realism and speed, based on soft contacts. Although it was proprietary software for several years, MuJoCo

was recently acquired by DeepMind and is now free to use and open source. Other notable physics engines include Bullet (Coumans, 2010), Box2D (Catto, 2011), Open Dynamics Engine (ODE) (Smith et al., 2005), and Dynamic Animation and Robotics Toolkit (DART) (Lee et al., 2018a). Most engines are highly optimized, using compiled languages such as C++ and are not differentiable. However, some have been specifically designed to be differentiable, such as Tiny Differentiable Simulator (TDS) (Heiden et al., 2021) and Brax (Freeman et al., 2021), allowing trajectory optimization via gradient descent.

Simulation models For continuous control environments, models describe the state of the world used by the physics engine. The controllable parts of models are often inspired by animals such as snakes, bipeds, and quadrupeds. When designing a new model, a large amount of time can be invested in fine-tuning bodies, geometries, sites, joints, actuators, sensors, and various hyperparameters such as friction, stiffness, and damping. Small differences in the design of a model can have a dramatic impact on the resulting policies. A well-designed model can already exhibit sophisticated motion without much actuation. Famous real-world examples include motorless walking robots and dead fishes swimming upstream. It should also be noted that while some realistic robot models exist, many popular models use non-realistic torque-controlled multiaxial joints.

Resets At the beginning of each episode, a reset of the environment is performed. In theory, this involves sampling the initial state from a distribution of states. For a walking task, this may involve placing a bipedal walker in an upright pose, while for a target-reaching task, the target location may be randomized. However, changing the state of the environment is not mandatory and some tasks may continue from the last state of an episode. The reset function typically returns the initial episode observation without a reward or a termination signal. A typical challenge when designing training infrastructures using distributed training is how to handle asynchronous resets and what to communicate to the agent.

Steps After receiving an action from the agent, the environment advances its state, following the dynamics of the underlying Markov Decision Process. In practice, the rules of a game or the forward function of a physics engine provide the next state from the previous state and an action.

Observations Traditional reinforcement learning algorithms often assume that the agent has access to Markov states. This is indeed the case in a number of toy

problems where states can be enumerated or with continuous control environments, where the joint values and velocities may be sufficient to fully describe the world. However, a large number of environments provide only partial observations of states. In particular, for Atari 2600 games, RGB frames corresponding to the screen observation are provided. To allow agents to deal with this partial observability, for example to determine the movement of objects, multiple RGB frames are typically stacked or a recurrent network such as an LSTM is used. Finally, instead of being composed of a single type of input, observations can also be a group of inputs of different nature, such as proprioception and first-person vision.

Actions To influence the environment, agents must produce actions in an action space, generally discrete and continuous. For discrete action spaces, agents must choose an element from a set of possible values. This is the case for Atari 2600 games where the combination of joystick directions and button presses defines the options available to the agent. Instead, continuous action spaces are defined by intervals of values in multiple dimensions. A vector in this space would typically represent the amount of torque to apply to the available motors.

Rewards The reward function provides a mapping from environment states and actions to values. In some cases, a reward may directly specify the desirability of a state, but more generally, it should be defined such that maximizing the sum of discounted rewards generates a policy solving the task. Designing a good reward function is difficult. It requires to reverse engineer a desired policy and can vary depending on the agent used. For example, for an agent capable of efficient exploration, a sparse reward may suffice, but for others, a shaped reward may be needed to provide a useful gradient for the agent's progress. A reward can also be used to penalize certain behaviors such as using too much torque in continuous control or to provide a constant penalty to create temporal pressure. In addition, some rewards can be generated internally by an agent. This can be used, for example, to stimulate exploration, but is not part of the environment per se.

Terminations When an agent encounters a terminal state, the current episode ends and a new episode can begin. The conditions for these terminations must also be designed. Examples include a walker falling to the ground or a goal being reached. Changing these terminations can have a significant impact on an agent's learning, as strict conditions around a desired behavior can reduce the problem complexity. It is important to note that a large majority of available environments also rely on time limits to terminate episodes.

Demonstrations In complex environments, finding an effective policy may be difficult, especially in sparse reward settings with long-term credit assignment, or with hard-to-control bodies such as humanoids. Current exploration techniques may not be sufficient, and even if a non-trivial policy is found, it may get stuck in a local optimum. For these reasons, the use of demonstrations has proven extremely valuable in a number of learning situations. In addition, demonstrations are often considered in robotics to demonstrate the task at hand in multi-task scenarios.

6.3 Elements of environment design in the thesis

1. In Chapter 4, a set of custom reaching environments inspired by Reacher-v1 with 3, 4, and 5 DOF is proposed. Furthermore, the continuous action spaces from Hopper-v1, Walker2d-v1 and Humanoid-v1 environments in OpenAI Gym are transformed into discrete action spaces via discretization of the action dimensions.

2. In Chapter 5, a set of randomly generated random mazes with goals to reach are proposed along Sokoban puzzles with a single box to push to a target. Furthermore, a modified version of Montezuma's Revenge and Super Mario All-Stars with the coordinate of the avatar extracted from the RAM are used.

3. In Chapter 7, motion capture tracking and complementary tasks are combined within a single Humanoid-based environment, switching between those tasks to help an agent learn reusable skills. Other challenging locomotion tasks are also used to demonstrate the reuse of learned skills.

4. In Chapter 8, a musculoskeletal model of an ostrich is proposed to study muscle-based actuation. It is based on anatomical data acquired on real ostriches and a dataset of motion capture clips is adapted to the model. Furthermore, tasks including motion capture tracking and neck control are proposed to facilitate the connection between biomechanics and reinforcement learning.

5. In Chapter 10, a clarification is proposed for the nature of time limit terminations found in popular environments. In particular, OpenAI Gym environments are modified to separate time limit terminations from environmental ones, and multiple toy problems are proposed to illustrate some of the discussed properties of time limits.

6. In Chapter 11, a set of differentiable environments based on a novel framework-agnostic library, called Ivy, is proposed. Some of those environments are adaptations of classic toy environments and allow trajectory optimization using gradient descent.

7. In Chapter 12, the recommendations from Chapter 10 are applied to environments from OpenAI Gym, dm_control and PyBullet and those environments are wrapped to obtain a uniform API and synchronous distributed training.

Chapter 7

Learning reusable motor primitives

This chapter is based on the paper "CoMic: Complementary task learning & mimicry for reusable skills" (Hasenclever et al., 2020) published at the ICML 2020 conference.

The videos are available at: https://youtu.be/JUZmchgYh7E, https://youtu.be/Ke6Ha4opYk8, and https://youtu.be/ZssTZpWro7Q.
The source code of the tasks is available at: https://github.com/deepmind/dm_control/tree/master/dm_control/locomotion.
The motion capture clips are adapted from the CMU mocap database, available at: http://mocap.cs.cmu.edu.

The main research questions are:

- How to learn reusable skills from mimicry?
- How to keep learning skills while learning to solve other tasks?
- Which choices of neural architecture and embedding spaces matter?

7.1 Introduction

Learning policies that can produce complex motor behavior for articulated, physically simulated bodies is a difficult challenge. Recent efforts have demonstrated that robust policies for certain locomotion behaviors can be learned from scratch in rich environments (Heess et al., 2017). However, this can be very data-inefficient and the quality of the resulting behavior may be lacking. To alleviate this problem, it is desirable to be able to repurpose previously learned skills and use prior knowledge to facilitate and speed up learning on subsequent tasks. Various efforts have sought to design hierarchical architectures that enable reuse of skills (Heess et al., 2016; Florensa et al., 2017a; Haarnoja et al., 2018a), often through factoring the problem into learning a high-level task-specific module and a reusable low-level controller.

While hierarchical architectures seem especially promising in multi-task rein-

forcement learning, discovering robust behaviors via standard exploration techniques remains difficult. The pre-trained skills could come from other, perhaps simpler, tasks (Riedmiller et al., 2018) but this requires careful task design and is difficult to scale to large numbers of tasks. Alternatively, the skills could be derived from expert demonstrations. In the case of humanoid control, these demonstrations can be acquired from the large amount of motion capture (mocap) data publicly available (Merel et al., 2018b; Peng et al., 2019). However, while very effective, this approach may prevent agents from solving tasks requiring other skills not well covered by the demonstrations.

In this work, we explore this space further, studying the trade-offs of various existing approaches for representing and transferring reusable skills. How well skills can be learned and transferred to new tasks depends on a number of factors, including the nature and number of the pre-training tasks, as well as the structure and capacity of the architecture, which determines the ability to represent, reuse, and generalize skills. In our experiments, we demonstrate that simpler architectures tend to perform better than specialized ones, and that while a modest amount of motion capture data is sufficient to solve challenging locomotion tasks, asymptotic performance improves with data size.

Furthermore, we propose a flexible framework to simultaneously learn to track motion capture clips and solve complementary tasks. Our results show that new skills can be discovered from additional tasks along with the ones from the reference dataset, and that their discovery is actually facilitated by the motion capture tracking compared to finding them from scratch. Furthermore, the resulting skill embedding space incorporates both the reference movements and those required to solve the complementary tasks.

7.2 Related work

7.2.1 Learning skills

In the context of continuous control, transferable motor skills can be obtained in various ways. A simple approach can be to manually design separate controllers for specific behaviors. However, if hand-designed controllers can be sequenced in novel ways, they provide limited flexibility. Contemporary efforts have therefore focused on learning-based approaches. A first class of learning-based approaches involves carefully creating sets of tasks whose mastery leads to the emergence of useful skills (Heess et al., 2016; James et al., 2018; Riedmiller et al., 2018; Hausman et al., 2018). The expectation is that the skills that emerge to solve any particular task may be

useful for other tasks in the set. A second class of approaches involves unsupervised learning of skills (Gregor et al., 2016; Florensa et al., 2017a; Warde-Farley et al., 2018; Eysenbach et al., 2018), generally based on maximizing the diversity of visited states. While these approaches require less engineering, they often depend upon the specification of strong priors on the dimensions that skills should affect, for example, by prioritizing the body position in the plane to incentivize locomotion. Moreover, finding a large number of diverse behaviors can lead to skills that are not relevant for later tasks. A third alternative is therefore to use imitation learning techniques to acquire skills from sources such as motion capture (Merel et al., 2017, 2018b) or even loosely structured "play" (Lynch et al., 2020).

Articulated humanoids are an example of a high-dimensional body that is particularly complex to control. Our present work is situated most closely among the various efforts to produce robust, natural, and reusable motor behavior for physically simulated humanoids. In this setting, approaches which generate behavior by tracking motion capture demonstrations tend to produce the most robust and realistic movements (Liu et al., 2010; Peng et al., 2018; Chentanez et al., 2018; Bergamin et al., 2019). In addition, there has long been recognition in this field that skills should be able to be repurposed for new tasks (Faloutsos et al., 2001; Liu & Hodgins, 2017). We build most directly on recent efforts involving large-scale tracking of motion capture data (Chentanez et al., 2018) as well as efforts to build reusable skills from these data (Merel et al., 2018b; Peng et al., 2019).

7.2.2 Controlling skills

A common architectural motif among recent approaches to represent skills is the use of an embedding space (Heess et al., 2016). This can be repurposed in the context of hierarchical control schemes where, for a new task, a high-level controller produces actions via the embedding space. Embedding vectors can be used to trigger either a specific plan to reach a goal (Lynch et al., 2020), a behavior for an entire episode (Wang et al., 2017; Eysenbach et al., 2018), a segment in an episode (Hausman et al., 2018), or a temporally-correlated behavior specified at every time step (Merel et al., 2018b). To make the latent space easy to control for a high-level controller, the distribution of latent vectors is generally regularized to be well distributed and smooth. This regularization can also be viewed from an information bottleneck perspective (Goyal et al., 2019, 2020). The resulting skills are usually frozen, and reuse is limited by the behaviors that can be triggered via the latent space (Haarnoja et al., 2018a).

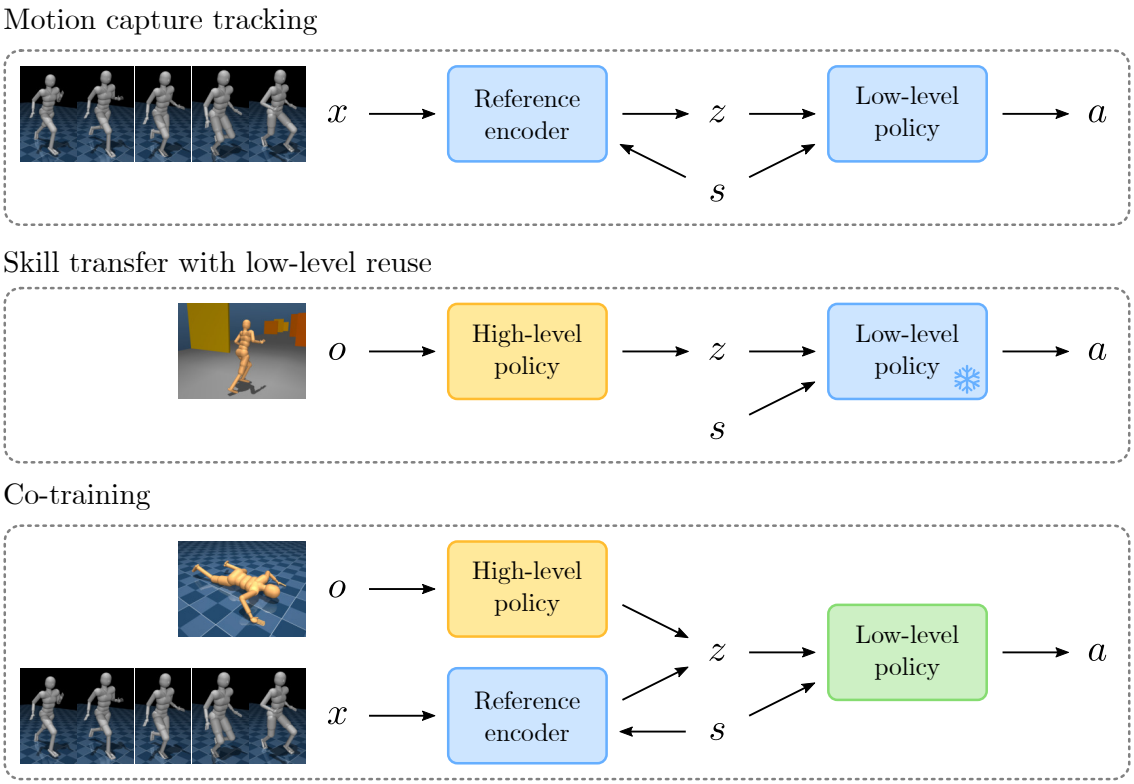


Figure 7.1: Methods considered to learn skills and use them. Top: Motion capture tracking using a reference x of future poses to follow, passed to a reference encoder with the body state s, that produces a latent encoding z, passed with s to a low-level policy producing an action a. After training, the low-level policy can be reused. Middle: Learning to solve a task by training a high-level policy that receives task observations o and that outputs an action in the embedding space. The low-level policy is frozen and cannot learn new skills. Bottom: Co-training where both motion capture tracking and complementary tasks are training the low-level policy simultaneously.

7.3 Methods

7.3.1 Learning reusable skills

Learning skills through mimicry Imitation of motion capture clips is used to learn low-dimensional skill embedding spaces that can then be reused to solve challenging control tasks. The architectures considered in this work use a reference encoder $\pi_{\text{HL}}(z_t|s_t, x_t)$, which, at time t encodes the desired future target states $x_t = (\hat{s}_{t+1}, \ldots, \hat{s}_{t+5})$ from the motion capture reference data and current state s_t into a stochastic embedding z_t, used to condition a low-level policy π_{LL}. This low-level policy also receives proprioceptive information about the state of the body as an input and produces a distribution over actions a_t (see Panel 1 in Figure 7.1).

Transferring skills to new tasks Having trained on the mocap tracking task, we can then transfer the low-level policy to a new task by reusing the learned embedding space as a new action space. To do this, we freeze the parameters of the low-level policy and learn a new high-level policy that outputs a latent embedding z_t at each time step (see Panel 2 in Figure 7.1). This high-level policy can take as inputs task observations o_t. Acting in the learned embedding space heavily biases the resulting behavior to the behaviors present in the motion capture data. This enables much more coherent exploration. For example, we observe that often, a randomly initialized high-level policy already results in naturalistic movements for an extended period of time while randomly initialized policies in the raw action space tend to fall immediately.

Joint training with complementary tasks On the other hand, the bias towards behaviors from the motion capture data can also hurt performance. Skills that are relevant for a transfer task but not well covered by the motion capture data may be impossible to learn. For example, getting up from the ground is underrepresented in many motion capture datasets and is a hard skill to learn but clearly desirable in transfer tasks. To tackle this problem, we propose a joint training approach: during the skill learning phase, we interleave the mocap tracking task with other, complementary, out-of-sample tasks that ensure that specific skills are well represented in the embedding space (see Panel 3 in Figure 7.1). Since complementary tasks do not share the same observation as the mocap tracking task and optimize for different rewards, we train a separate high-level policy acting in the learned embedding space but use the same low-level policy. This approach forces the embedding space to represent behaviors induced by the complementary tasks.

Multi-task RL The proposed approach to learning reusable skills relies on multi-task reinforcement learning, alternating tracking tasks with solving complementary ones. This conceptually simple approach contrasts with the complex training pipeline originally used for learning neural probabilistic motor primitives (NPMPs) (Merel et al., 2018b). The latter approach required the learning of hundreds of independent policies, one for each short clip segment (up to 200 steps), storing the sequences of observations and actions in an offline dataset, and distilling this dataset into an encoder-decoder architecture via supervised learning.

7.3.2 Neural network architectures

Reference encoder For the reference encoder, we concatenate the reference observations for the next 5 timesteps, proprioceptive observations and the sampled

latent z_{t-1} from the previous timestep. This combined observation is fed into an MLP with 2 hidden layers with 1024 hidden units each.

Skills control We restrict ourselves to architectures with a bottlenecked latent space (Hausman et al., 2018; Merel et al., 2018b) that serves as a new action space in transfer tasks. To facilitate reuse, the latent space is regularized using a Kullback–Leibler (KL) divergence penalty between the latent distribution and a standard normal distribution $\beta KL(\pi_{\mathrm{HL}}(z_t|s_t, x_t)\|\mathcal{N}(0, I))$, where the coefficient β controls the strength of the KL regularization. For the mixture and product architectures, the high-level policy acts in the space of primitive mixture weights or exponents.

Policy networks It is a priori unclear which policy architectures could allow an effective transfer of skills. The simplest choice is a monolithic feedforward architecture, but other hierarchical architectures might introduce additional structure by organizing the low-level policy into primitives. Two recent representative examples are mixture distributions (Wulfmeier et al., 2019) and product distributions (Peng et al., 2019). The promise of such modular architectures is compositionality: different primitives could specialize to different skills, which can then be flexibly composed by varying the primitive weights. Whether this hope is realized remains unclear and is likely highly task-dependent. Modularity is another possible benefit of mixture or product architectures. Both potentially allow the addition of new primitives and importantly allow primitives to specialize, which may help represent wide ranges of skills. Another plausible alternative is a recurrent low-level policy which can model temporal correlations in behavior and allows for temporally extended "default" behaviors.

Guided by these considerations, we compare four representative network architectures for the low-level policy in our experiments: a fully connected neural network (MLP), a recurrent neural network (LSTM), a mixture of Gaussians and a product of Gaussians.

The mixture distribution with C components is given by:

$$\pi_{\mathrm{LL}}(a_t|s_t, z_t) = \sum_{i=1}^{C} w_i(s_t, z_t)\phi(a_t|\mu_i(s_t), \sigma_i(s_t))$$

where $\phi(\cdot|\mu, \sigma)$ is the density of a normal distribution with diagonal covariance and with mean μ and standard deviation σ.

Similarly, the (unnormalized) product distribution is given by:

$$\pi_{\mathrm{LL}}(a_t|s_t, z_t) \propto \prod_{i=1}^{C} \phi(a_t|\mu_i(s_t), \sigma_i(s_t))^{w_i(s_t,z_t)}$$

For Gaussian factors, the resulting product distribution is Gaussian with mean and variance given analytically as function of w_i, μ_i, σ_i. Following Wulfmeier et al. (2019) and Peng et al. (2019), we do not provide the latent embedding z_t to the primitives to encourage specialization. In our experiments, both the mixture and the product policy networks have a shared torso followed by per-component one hidden layer MLPs. To keep the capacity of the low-level architectures similar, we choose networks with roughly similar number of parameters. We found that adding KL regularization made the training of product distributions much more stable.

In the case of the MLP low-level controller, we also use an additional linear pathway from the input to the output of the MLP. We found that this change slightly improved results but made no difference for other low-level architectures. We used linear layers on top of the resulting representations to produce mean and log standard deviation for stochastic latent embedding.

Value function For the value function used in the mocap tracking task, we concatenate the reference observations for the next 5 timesteps, proprioceptive observations, as well as a 30-dimensional learned clip embedding. This is fed into an MLP with 3 hidden layers with 1024 hidden units each. We use layer normalization and predict a value for each reward component separately on top of this representation.

7.3.3 Physics simulation

In this work, we perform our architecture comparisons and demonstrate our joint training approach in the context of humanoid continuous motor control tasks. Because motor control of high-dimensional, physically simulated bodies is difficult, it can be efficient to reuse skills. However, reuse is only helpful if previously acquired skills are relevant for subsequent challenges. To that end, given a distribution of reference motions, we consider tasks that we expect can be solved by reusing the skills found in the motion capture dataset as well as tasks which cannot. All tasks involve simulated physics using MuJoCo (Todorov et al., 2012). We use a humanoid body adapted from the "CMU humanoid" available in dm_control (Merel et al., 2018a). We adjusted limb lengths, masses, and dynamic properties of the body to make it more consistent with an average human. The humanoid is controlled with a simulation timestep of 5ms and a control timestep of 30ms (6 action repeats). See

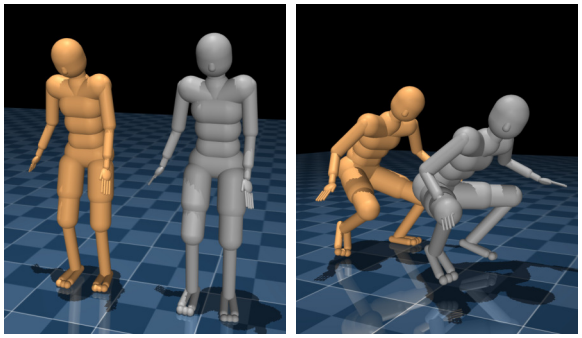


Figure 7.2: Examples of motion capture tracking with physically simulated humanoid (bronze) and offset mocap reference pose (grey). Note how the imperfect reference pose self-intersects.

Figure 7.2 for an illustration of the humanoid body in the context of our tracking task.

7.3.4 Motion capture tracking task

The primary task we employ to learn skills in this work is the multi-clip tracking task that is broadly similar to others used in prior work on motion capture tracking (Peng et al., 2018; Merel et al., 2018a; Chentanez et al., 2018; Peng et al., 2019). The important aspects of the tasks are initialization, observations, termination conditions, and reward function.

Initialization At the start of each episode, we randomly select a starting frame from all frames in the underlying set of clips (excluding the last 10 frames from each clip). At the beginning of each episode, the humanoid is initialized to the target pose in the selected frame.

Observations The agent receives both proprioceptive observations as well as information about the target pose in the active mocap clip. The proprioceptive observations are the joint angles, joint angular velocities, velocimeter observation, gyroscope observation, the end effector positions, the "up" direction in the frame of the humanoid, the actuator state, as well as touch sensors on the hands, fingers and feet and torque sensors in both shoulders. The observations about the reference poses consist of the relative target positions and orientations of different body parts in the local frame. We provide the agent with target poses $\{\hat{s}_{t+1}, \ldots, \hat{s}_{t+5}\}$ for the next 5 timesteps, similarly to Chentanez et al. (2018) and Merel et al. (2018b).

Reward and termination The reward function captures how close the state of the simulated humanoid is to the respective reference. Similar to previous work

(Peng et al., 2018; Merel et al., 2018a), we found the results to be relatively sensitive to the reward function. Our reward function contains five different terms:

$$r = \frac{1}{2} r_{\text{trunc}} + \frac{1}{2} \left(0.1 r_{\text{com}} + r_{vel} + 0.15 r_{\text{app}} + 0.65 r_{\text{quat}}\right)$$

The first reward term r_{trunc} penalizes deviations from the reference in joint angles and the Euclidean position of a set of 13 different body parts:

$$r_{\text{trunc}} = 1 - \frac{\varepsilon}{\tau} \quad \text{with} \quad \varepsilon = \|b_{\text{pos}} - b_{\text{pos}}^{\text{ref}}\|_1 + \|q_{\text{pos}} - q_{\text{pos}}^{\text{ref}}\|_1$$

where b_{pos} and $b_{\text{pos}}^{\text{ref}}$ correspond to the body positions of the simulated humanoid and the mocap reference and q_{pos} and $q_{\text{pos}}^{\text{ref}}$ correspond to the joint angles. This reward term is linked to the termination condition of our tracking task. Given a termination threshold τ, we terminate an episode if $\varepsilon > \tau$. Note that this ensures that $r_{\text{trunc}} \in [0, 1]$. We found that including this termination condition and the coupled reward speeds up training on larger clip sets but does not by itself lead to visually appealing tracking behavior. We use $\tau = 0.3$ in our experiments.

The second reward term is similar to the objective proposed in Peng et al. (2018) with terms penalizing deviations of the center of mass, joint angle velocities, end effector positions, and joint orientations with different weights. The first term r_{com} penalizes deviations of the center of mass:

$$r_{\text{com}} = \exp\left(-10 \|p_{\text{com}} - p_{\text{com}}^{\text{ref}}\|^2\right)$$

where p_{com} and $p_{\text{com}}^{\text{ref}}$ are the positions of the centre of mass of the simulated humanoid and of the mocap reference, respectively.

The second term r_{vel} penalizes deviations from the reference joint angle velocities:

$$r_{\text{vel}} = \exp\left(-0.1 \|q_{\text{vel}} - q_{\text{vel}}^{\text{ref}}\|^2\right)$$

where q_{vel} and $q_{\text{vel}}^{\text{ref}}$ are the joint angle velocities of the simulated humanoid and of the mocap reference, respectively.

The third term r_{app} penalizes deviations from the reference end effector positions:

$$r_{\text{app}} = \exp\left(-40 \|p_{\text{app}} - p_{\text{app}}^{\text{ref}}\|^2\right)$$

where p_{app} and $p_{\text{app}}^{\text{ref}}$ are the end effector positions of the simulated humanoid and of the mocap reference, respectively.

Finally, r_{quat} penalizes deviations from the reference in terms of the quaternions describing the joint orientations:

$$r_{\text{quat}} = \exp\left(-2 \|q_{\text{quat}} \ominus q_{\text{quat}}^{\text{ref}}\|^2\right)$$

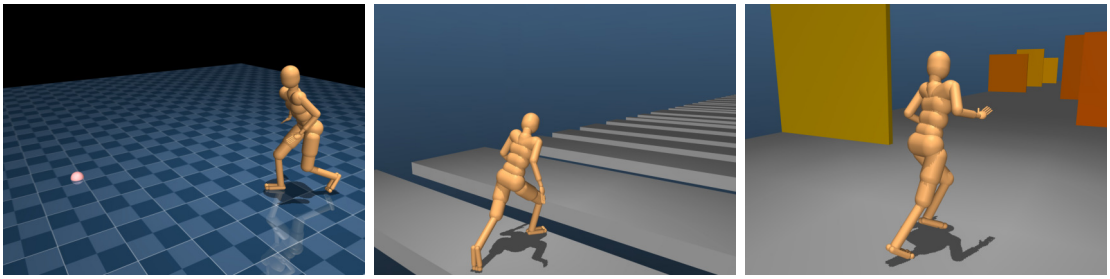


Figure 7.3: Environments used to evaluate the transfer of trained low-level policies to challenging locomotion tasks: (left) go-to-target (center) gaps (right) walls.

where $\ominus$ denotes quaternion differences and q_{quat} and $q_{\text{quat}}^{\text{ref}}$ are the joint quaternions of the simulated humanoid and of the mocap reference, respectively.

7.3.5 Transfer tasks

To evaluate the reusability of the learned low-level policies, we use three challenging locomotion tasks from dm_control (Merel et al., 2018a; Tassa et al., 2020). Locomotion tasks form a natural, well-motivated test bed for humanoid skills that is within reach of current methods. In all tasks we initialize the humanoid using suitable motion capture pose. These tasks are illustrated in Figure 7.3.

Go-to-target We consider a sparse go-to-target task. The simulated humanoid is randomly initialized in an arena. The task requires locomoting to a random target location. The agent is rewarded for being within 1 meter of the target location for 10 steps after which the target randomly moves to a new location. The task terminates after 25 seconds.

Gaps and walls obstacle courses We consider two challenging obstacle courses to be solved from first-person visual observations. In both tasks, the agent is rewarded for achieving a target root velocity of 3m/s in the direction of the corridor. We use the default reward function provided by the task: it increases linearly from 0 at 0m/s to 1 at 3m/s and decreases linearly to 0 at 6m/s. In both cases, the task is terminated after 45 seconds, and we use a corridor length of 100m and a corridor width of 10m. For the gaps task, the length of the gaps is randomized between 0.75m and 1.25m, and the length of the platforms between gaps is randomized between 0.3m and 2.5m. For the walls task, the distance between walls is set to 5m, they are randomly placed on either side, their length is randomized between 1m and 7m, their height is randomized between 2.5m and 4m, and their color is also randomized.

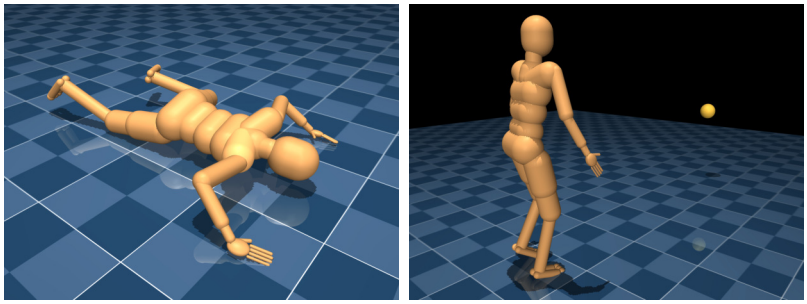


Figure 7.4: Environments used for joint training. Some skills are not covered well by most motion capture datasets. To obtain a rich skill space we propose to train jointly with complementary tasks such as getting up (left) and catching a ball (right).

7.3.6 Complementary tasks

Some motor skills may not be available, or may be underrepresented in the motion capture dataset. For example, the CMU motion capture database has relatively few clips involving getting up from the ground, compared to walking. After training a policy that is capable of tracking a subset of the motion capture library, the reusable low-level controller may not provide an advantage on a task that requires the humanoid to get up from the ground. Rather, the low-level controller may even bias the learning away from such movements. Consequently, we also consider two representative *out-of-sample* tasks that involve movements that were either underrepresented or absent from our tracking reference data. These tasks are illustrated in Figure 7.4.

Get-up and stand The first task we consider, is a get-up and stand task. The humanoid is initialized in a large variety of poses, about 5% lying on the ground and 95% floating slightly above the ground in a standing pose, which induces falling in a variety of different ways. The reward function is $\exp(-(h - h_{\text{target}})^2)$, where h is the head height of the humanoid and h_{target} is the target height corresponding to standing. The task terminates after 8 seconds. For perfect task performance, the humanoid has to both avoid falling when initialized in a standing pose and learn to get up from the ground when initialized lying down.

Ball catching The second task involves catching a ball thrown toward the humanoid. This task is similar to the one considered by Merel et al. (2020). In this task, the humanoid is initialized in a standing pose. The ball is initialized in mid-air with a random velocity component propelling it upwards and towards the humanoid. The size and mass of the ball, as well as its angular velocity are also randomly sampled across episodes. Episodes terminate after 6 seconds or if the ball makes contact with

the ground or hits the humanoid's head. In the event of failure, the agent receives a reward of -1. Furthermore, the agent receives a reward of $+0.01$ for each hand touching the ball with the front and a penalty of -0.01 for touching the ball with the back. In addition, we use a small shaping reward to encourage standing up. The reward is identical to the one used in the get-up and stand task, but scaled to a maximum of 0.01 per timestep. Upon catching the ball, there is a small shaping reward incentivizing the humanoid to walk forward towards a target.

7.4 Experiments

Training infrastructure Unless otherwise stated, we use V-MPO (Song et al., 2019), an on-policy variant of Maximum a Posteriori Policy Optimization (Abdolmaleki et al., 2018). In the case of mixture networks, we use an additional KL constraint on the distribution of mixture components similar to Wulfmeier et al. (2019). To train the critics, n-step returns are used. In the V-MPO E-Step we use the top 50% of the advantages as suggested by Song et al. (2019) and all Lagrange multipliers in V-MPO were initialized by default to 1. We use the Adam optimizer with a learning rate of 10^{-4}, a batch size of 128, an unroll length of 32 and a discount factor of 0.95. We use a distributed infrastructure with a single learner on a 1x1 TPUv2 chip (with 2 cores) (Google, 2018) and use 4000 actor processes to produce environment interactions for the learner. A single experiment takes about 2-3 days.

Transfer experiments For the transfer tasks, we use SVG(0) (Heess et al., 2015) with the Retrace off-policy correction (Munos et al., 2016) and shared observation encoders. Small MLPs are used for the proprioceptive observations and the non-visual task observations in the go-to-target, get-up, and ball catching tasks. For the visual observations in the walls and gaps tasks, we use a small ResNet. The observation encodings are concatenated and processed by an LSTM with 128 units that is shared between the policy and the state-action value function. On top of this LSTM, an MLP with 256 hidden units is used for the state-action value function, and another LSTM with 128 units followed by linear layers to produce the high-level policy.

Data regimes Different architectures inject different inductive biases and the effect of these model choices may be more pronounced in small data regimes. Hence, to study this effect we define four different data regimes: a two minute set of clips containing walking behaviors with various turns ("walking"), a two minute set of clips containing walking behaviors as well as running and jumping ("running"), a

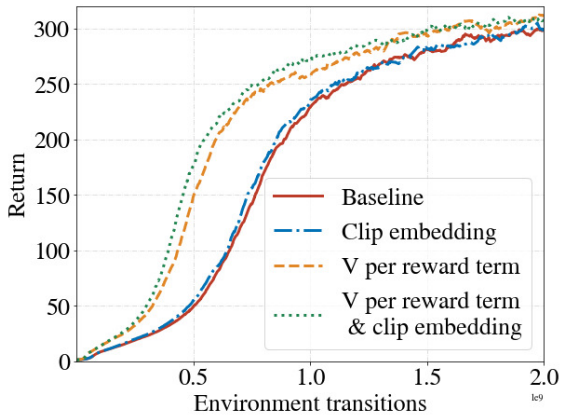


Figure 7.5: Tracking performance of the considered tricks. Learning separate value functions per reward component speeds up training, but does not affect asymptotic performance. The figure shows results for an MLP low-level on the "locomotion" dataset with $D = 60$ and $\beta = 1 \times 10^{-4}$.

40 minute set of locomotion clips ("locomotion") as well as a 220 minute set of clips with a wide range of locomotion behaviors and hand movements ("large"). All motion capture data was obtained from the CMU mocap database.

7.4.1 Scaling up motion capture tracking

We find that several factors affect how well a policy can be trained on multiple clips. Firstly, we found on-policy reinforcement learning easier to scale to large motion capture datasets. Secondly, better learning of the value function improved learning speed. Specifically, we explored two different approaches to improve value learning. We explored giving the value function access to a one-hot encoding of the reference clip id. Secondly, we exploit the linear nature of the reward function and learn a separate value function per reward term (Van Seijen et al., 2017) (see Figure 7.5). We found that learning a separate value function per reward term significantly speeds up learning whereas conditioning on the clip id only results in a modest additional improvement. We use both of these improvements in all further experiments. In line with previous work (Peng et al., 2018; Merel et al., 2018a), we observed that the reward function needs to be chosen carefully for best results although it is worth noting that concurrent work (Abdolmaleki et al., 2020) alleviates this difficulty.

7.4.2 Latent space dimensionality and KL regularization

When training reusable low-level controllers, an important choice is the size of the learned latent space. On the one hand, choosing a smaller size reduces the action

β	1×10^{-5}	1×10^{-4}	2×10^{-4}	5×10^{-4}
$D = 20$	0.69	0.69	0.69	0.62
$D = 40$	0.69	0.69	0.70	0.59
$D = 60$	0.70	0.69	0.70	0.62

Table 7.1: Average tracking reward per step for various latent space dimensionalities D and KL regularization strengths β after 6×10^9 environment transitions processed.

space in transfer tasks and could lead to faster learning. On the other hand, a smaller latent space can represent fewer behaviors and might impair transfer performance. In addition, training the low-level controller on the mocap tracking task is likely to be harder, the smaller the learned latent space. Similar considerations apply to regularization in the learned latent space. Stronger regularization should be beneficial in transfer tasks since random exploration in the latent space at the beginning of training will be closer to the distribution of embeddings encountered during tracking, leading to more coherent exploration. However, too strong a regularization might impair the tracking performance and reduce the number of behaviors represented in the latent space.

Motion capture tracking To study these questions we trained some MLP low-level controllers with 2 hidden layers of 1024 hidden units followed by linear layers to produce mean and log standard deviation. The inputs are a concatenation of the latent embedding z_t and the proprioceptive observations. The initial values of the Lagrange multipliers were $\epsilon = 0.1$, $\epsilon_\mu = 0.1$ and $\epsilon_\Sigma = 10^{-5}$. We use the "locomotion" clip set with latent space dimensions $D = 20, 40$ and 60 and a range of KL regularization strengths β. The tracking performance is shown in Table 7.1. It remains roughly constant for different values of D and for a range of different values of β and begins to degrade slightly for $\beta = 5 \times 10^{-4}$. Even higher values of β lead to substantially reduced performance.

Transfer tasks Next, we evaluate the trained low-level controllers on our transfer tasks. Figure 7.6 shows the performance on the go-to-target task. Across values of D, we find that the most highly regularized low-level policies encounter rewards fastest. This reflects the initial exploration behavior with a randomly initialized high-level controller. For the most regularized controller the initial behaviors is to take a few steps while less regularized policies lead to almost immediate falling. Note that for high values of D, the regularization strength has a particularly strong effect on transfer learning speed. The transfer result to the walls and gaps tasks are shown in Table 7.2 and Table 7.3. On the walls task, all low-level controllers perform quite

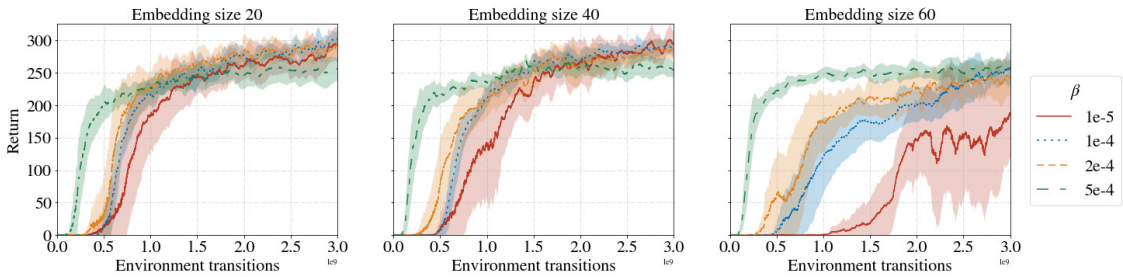


Figure 7.6: Go-to-target transfer performance for various latent space dimensionalities D and KL regularization strengths β (averaged over 5 runs). More highly regularized models encounter rewards faster but achieve slightly lower asymptotic performance.

β	1×10^{-5}	1×10^{-4}	2×10^{-4}	5×10^{-4}
$D = 20$	514 ± 6	550 ± 5	551 ± 5	578 ± 5
$D = 40$	536 ± 5	532 ± 6	520 ± 5	576 ± 6
$D = 60$	527 ± 5	447 ± 5	614 ± 7	628 ± 5

Table 7.2: Mean return with standard errors on the gaps transfer task for various embedding space sizes D and KL regularization strengths β. The returns are averaged over the logged rewards of a single run between 1.45×10^9 and 1.5×10^9 environment transitions processed.

β	1×10^{-5}	1×10^{-4}	2×10^{-4}	5×10^{-4}
$D = 20$	576 ± 4	635 ± 4	643 ± 4	817 ± 3
$D = 40$	641 ± 5	591 ± 4	621 ± 4	785 ± 4
$D = 60$	647 ± 4	524 ± 4	592 ± 4	788 ± 4

Table 7.3: Mean return with standard errors on the walls transfer task for various sizes of embedded spaces D and KL regularization strengths β. The returns are averaged over the logged rewards of a single run between 0.95×10^9 and 1×10^9 environment transitions processed.

well but the most highly regularized models perform best. On the gaps task the picture is less clear, but there is slight trend for more highly regularized models to perform best.

7.4.3 Architecture comparison

In this experiment we compared a number of different low-level architectures across four different data regimes ranging from two minutes worth of motion capture data to 220 minours. For all architectures, we optimized the V-MPO hyperparameters separately on the "locomotion" clip set (40 min) based on downstream transfer performance and kept them the same across all other data regimes. We chose a

low-level architecture with a similar number of parameters as the MLP low-level.

MLP For the low-level MLP, we use the same architecture as in the KL regularization and embedding size experiment with an embedding size of 60 and a KL regularization strength of 5×10^{-4}. The initial values of the Lagrange multipliers were $\epsilon = 0.1$, $\epsilon_\mu = 0.1$ and $\epsilon_\Sigma = 10^{-5}$.

LSTM For the LSTM experiments we use two stacked LSTM with 384 and 256 units, a recurrent value function instead of the feedforward value function described above, an embedding size of 60, and a KL regularization strength of 5×10^{-4}. The initial values of the Lagrange multipliers were the same as those of the MLPs.

Mixture and Product For the mixture and product architectures, we use a shared torso with two hidden layers of size 512, followed by separate MLPs with 256 hidden units per primitive, and 5 primitives. In preliminary experiments, we also explored using more primitives without substantially different results. The initial values of the Lagrange multipliers for the mixture architecture were $\epsilon = 0.1$, $\epsilon_\mu = 0.1$, $\epsilon_\Sigma = 10^{-3}$, and $\epsilon_{\text{discrete}} = 10^{-3}$. The initial values of the Lagrange multipliers for the product architecture were $\epsilon = 0.1$, $\epsilon_\mu = 0.1$, and $\epsilon_\Sigma = 10^{-3}$. For mixture and product architectures, directly regularizing the space of mixture weights or exponents with a KL penalty does not seem sensible. Thus, for both mixture and product low-level architectures, we experimented with different ways to include regularization in the architecture. In addition to the default implementation without a latent bottleneck or KL regularization, we also experimented with a latent bottleneck of dimensionality 60. In this case, the latent produced by the reference encoder is concatenated with the proprioceptive information and fed through a two-layer MLP with 400 hidden units each to produce the mixture weights or exponents, respectively. For each data regime, we compare the default implementation without KL regularization, and version with a latent bottleneck and different regularization strengths. We also tried explicitly initializing the different components far away from each other to encourage the use of different primitives by using a fixed bias in the mean of each component. Specifically, we explored biases of $-0.66, -0.33, \ldots, 0.66$ for the 5 primitives. We report the best performing architectures in Table 7.4. We note that we found training a product architecture without additional regularization very unstable.

The performance of different architectures on the motion capture tracking task is shown in Figure 7.7. Across all data sets, we found that LSTM and MLP low-level policies tended to be easier to train and outperform mixture and product low-level policies. While the asymptotic performance on the two smallest motion capture

	Best mixture architecture
Walking (2 min)	no bottleneck, bias init
Running (2 min)	no bottleneck, bias init
Locomotion (40 min)	no bottleneck, bias init
Large (220 min)	bottleneck $\beta = 5 \times 10^{-4}$ and bias init
	Best product architecture
Walking (2 min)	bottleneck $\beta = 5 \times 10^{-4}$
Running (2 min)	bottleneck $\beta = 5 \times 10^{-4}$
Locomotion (40 min)	bottleneck $\beta = 5 \times 10^{-4}$
Large (220 min)	bottleneck $\beta = 1 \times 10^{-4}$

Table 7.4: Best performing modular architectures for each data regime.

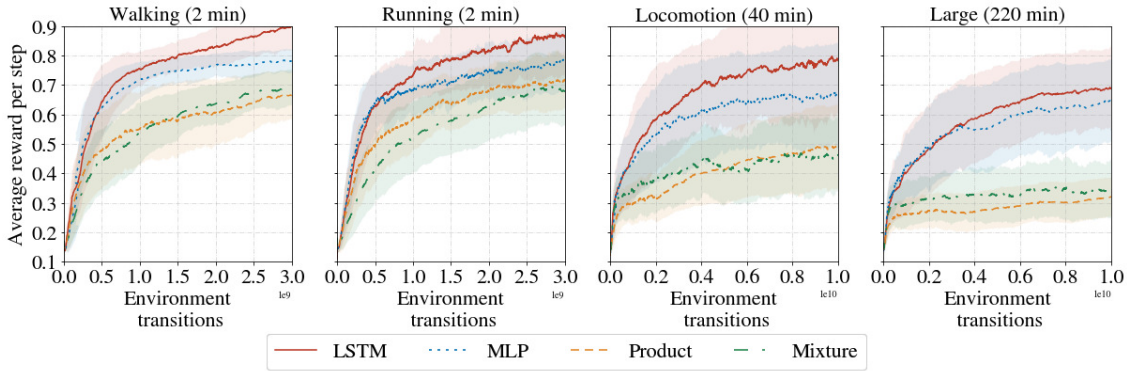


Figure 7.7: Architecture comparison across various data regimes. While modular architectures were competitive in small data regimes we found it very challenging to scale them up to large motion capture datasets.

datasets is similar, we found it difficult to achieve good tracking performance on the larger motion capture datasets with product and mixture low-level policies. One possible reason is the fact that, following Peng et al. (2019) and Wulfmeier et al. (2019), the primitives are not conditioned on the latent variable to encourage specialization, making it harder to learn on a large motion capture datasets. The LSTM low-level policy performs best on the mocap tracking task, since it can model the temporal structure in the behavior and help overfit to difficult reference clips. See Video 1 for examples of the tracking performance of the LSTM architecture on our largest dataset.

Next, we evaluated the trained low-level policies on the locomotion transfer tasks (see Table 7.5, Table 7.6 and Table 7.7). Across all transfer tasks, we found the MLP low-level policy to perform best. The LSTM performed less well, perhaps reflecting the fact that it can model temporal structure independent of the latent space, limiting its controllability. In our setting, the mixture and product policies

Go to target	Walking (2 min)	Running (2 min)	Locomotion (40 min)	Large (220 min)
MLP	$\mathbf{151 \pm 6}$	$\mathbf{213 \pm 6}$	$\mathbf{261 \pm 4}$	$\mathbf{280 \pm 7}$
LSTM	19 ± 1	30 ± 1	181 ± 4	264 ± 5
Mixture	19 ± 1	84 ± 2	100 ± 3	177 ± 4
Product	73 ± 3	13 ± 1	142 ± 3	154 ± 3

Table 7.5: Mean return with standard errors on the go-to-target transfer task for different architectures and data regimes. The returns are averaged over the logged rewards between 2.9×10^9 and 3.0×10^9 environment transitions processed and three runs. Across all data regimes the MLP low-level policy performs best.

Gaps	Walking (2 min)	Running (2 min)	Locomotion (40 min)	Large (220 min)
MLP	$\mathbf{461 \pm 8}$	$\mathbf{500 \pm 8}$	$\mathbf{628 \pm 5}$	$\mathbf{556 \pm 3}$
LSTM	77 ± 1	217 ± 3	261 ± 2	266 ± 2
Mixture	144 ± 1	421 ± 4	506 ± 4	433 ± 3
Product	164 ± 1	216 ± 1	174 ± 2	417 ± 6

Table 7.6: Mean return on the gaps transfer task for different architectures and data regimes. The returns are averaged over the logged rewards of one run between 1.4×10^9 and 1.5×10^9 environment transitions processed. Across all data regimes the MLP low-level policy performs best.

Walls	Walking (2 min)	Running (2 min)	Locomotion (40 min)	Large (220 min)
MLP	$\mathbf{626 \pm 5}$	$\mathbf{569 \pm 5}$	$\mathbf{807 \pm 6}$	622 ± 5
LSTM	152 ± 2	368 ± 5	660 ± 7	$\mathbf{725 \pm 6}$
Mixture	159 ± 2	351 ± 4	512 ± 11	679 ± 10
Product	391 ± 5	264 ± 4	638 ± 7	647 ± 6

Table 7.7: Mean return on the walls transfer task for different architectures and data regimes. The returns are averaged over the logged rewards of one run between 0.95×10^9 and 1×10^9 environment transitions processed. The MLP low-level policy performs best in most settings.

performed worse than the MLP across all data regimes despite considerable tuning effort. See Video 2 for examples of good performance on the locomotion transfer tasks.

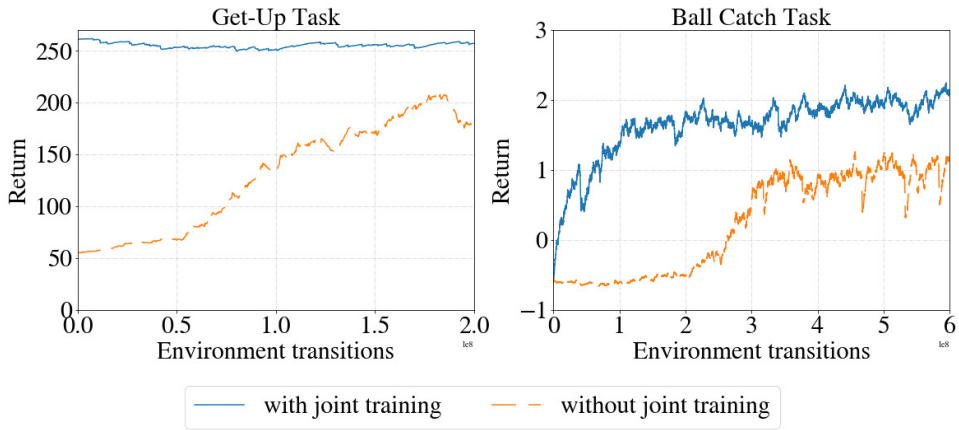


Figure 7.8: Performance on the complementary tasks with and without joint training. In the case of get-up and stand, the jointly trained low-level policy almost instantly solves the task. Even a randomly initialized high-level policy will rarely fall, and sometimes suddenly raise itself from the ground.

7.4.4 Joint training with complementary tasks

We consider joint training with two tasks that are not well covered by the motion capture data: getting up from the ground (present in only a few short clips) and catching a ball (completely absent). For these experiments, we train a low-level policy on the largest motion capture clip set and an additional task. For joint training with the get-up and stand task, at the beginning of each episode, the mocap tracking task is selected with 90% probability and the get-up task is selected with 10% probability. The ball-catching task is sampled 20% of the time. We use an MLP low-level architecture and per-task high-level controllers and value functions. For the mocap tracking task, we use the default architecture, and the same architectures for high-level controllers and value functions in both the tracking task and the complementary tasks. We did not learn separate value function per reward term in these experiments. We find that in both settings, joint training slightly degrades the tracking performance, but the performance on transfer locomotion tasks is unchanged. We also investigate the transfer performance on the joint training tasks (see Figure 7.8). In the case of get-up and stand, the jointly trained low-level policy almost instantly solves the task. Even a randomly initialized high-level policy will rarely fall and sometimes spontaneously get up from the ground in a visually human-like fashion. The low-level policy without joint training learns much more slowly and only manages to not fall but not to get up from the ground. Similarly, in the case of ball catching, the task is solved faster with joint training, and with higher performance, probably due to the various arm movements present in the dataset. See Video 3 for examples of behaviors on the complementary tasks.

7.5 Discussion

This work explored how to learn diverse, reusable skills via imitation and joint training on complementary tasks. Furthermore, we studied which architectures facilitate effective skill reuse. Prior work on learning reusable skills from large motion capture datasets showed that it is possible to train reusable skills combining reinforcement learning and distillation (Merel et al., 2018b, 2020). In these works, expert policies are trained to track short segments of individual mocap clips. Expert trajectories are then distilled into a single model in a supervised step with a low-level policy that can be transferred to new tasks. In contrast, in this work we train the low-level policy directly via RL on large sets of mocap data. This is significantly simpler, faster and cheaper than the two-stage approach taken by Merel et al. (2018b) and Merel et al. (2020). Furthermore, focusing on reinforcement learning provides greater flexibility in terms of the training setting.

We proposed a framework for training reusable skills on large corpora of motion capture data. We compared a variety of different embedding space sizes and regularization strengths. We found that transfer results were relatively insensitive to the embedding space size and that strong regularization mostly helps transfer on sparse reward tasks. We systematically compared a variety of different network architectures across a range of different data regimes and found that a simple feedforward architecture worked best. In particular, we found that, despite considerable effort, it was difficult to scale modular architectures with several primitives to large motion capture datasets. Indeed, even in the low-data regime, a simple feedforward low-level policy outperformed modular architectures. This result is at odds with results by Peng et al. (2019) and we speculate that a reason for this discrepancy is the more challenging nature of our tasks and the more realistic humanoid body. We leave a further investigation to future work. We found that the tasks we considered could be solved with reasonable performance using only about two minutes worth of motion capture data but asymptotic performance improved with the size of the underlying motion capture dataset.

In addition, we showed that it is possible to jointly train on complementary tasks, providing an easy way to ensure that skills not well covered by motion capture data are well represented in the learned embedding space. Future work could include leveraging richer environments than considered in this work, or combining imitation approaches with unsupervised skill discovery. Ultimately, we believe that the general approach of learning skills jointly through imitation and complementary tasks is very promising, offering the best of both worlds.

Chapter 8

Building and controlling musculoskeletal models

This chapter is based on the paper "OstrichRL: A musculoskeletal ostrich simulation to study bio-mechanical locomotion" (La Barbera et al., 2021) presented at the NeurIPS 2021 deep RL workshop.

The videos are available at: https://sites.google.com/view/ostrichrl.
The source code is available at: https://github.com/vittorione94/ostrichrl.

The main research questions are:

- How to create a realistic musculoskeletal model in MuJoCo?
- Can we control it via reinforcement learning?
- Can we obtain visually and physiologically realistic movements?

8.1 Introduction

In nature, vertebrates produce movement by the contraction of skeletal muscles that pull on the bones, creating torque at the joints. Understanding muscles is of interest in many different fields. First, from a biological point of view, it is desirable to understand how animals use their bodies to perform various behaviors. Muscles are also relevant in computer graphics to obtain accurate skin deformations in virtual characters and to produce more natural-looking gaits for films and video games (Angles et al., 2019; Abdrashitov et al., 2021; Modi et al., 2021). In sports medicine, musculoskeletal simulations can be used to understand sports injuries (Bulat et al., 2019) or create effective training and recovery strategies (Coste et al., 2017). Unfortunately, in robotics, with the exception of soft robots, muscles have been studied relatively little despite their interesting energetic properties and the sophisticated movements they enable (Wang et al., 2021; Cotton et al., 2012).

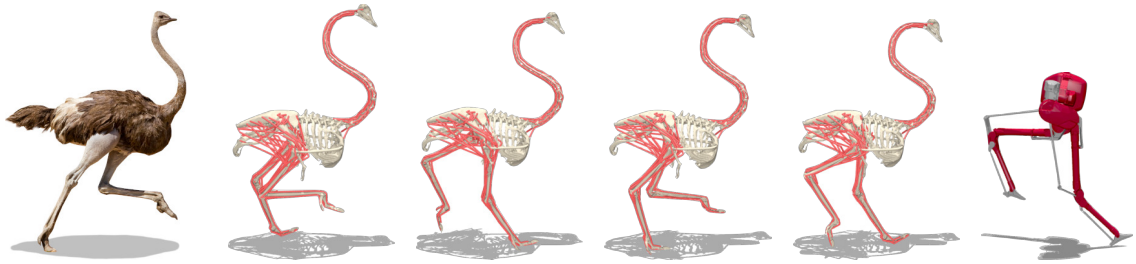


Figure 8.1: The proposed musculoskeletal model is based on data collected from real ostriches, while the Cassie robot provides a robotic approximation of the morphology of an ostrich.

Biomechanics is the study of how biological systems produce motion. Biological motion is affected by many factors such as the musculoskeletal geometry, internal states of the muscles, and force-length-velocity curves. Understanding how complex biological systems such as humans and other animals use their muscles to move is quite challenging. Obtaining data from living animals is difficult and not neutral to animal welfare. To overcome this problem, biomechanics researchers often use computer-based simulations combined with numerical optimization to estimate how muscles are used. However, most of the research that uses optimization techniques does not leverage recent progress made in reinforcement learning.

In this work, we propose a new ostrich musculoskeletal model, illustrated in Figure 8.1, that uses a fast physics engine with reinforcement learning tasks. Ostriches are interesting to study because they are fast, economical bipedal runners (Alexander et al., 1979). To the best of our knowledge, we are the first to apply RL to such a realistic animal model, solving locomotion tasks. With the proposed open-source computational tools, we hope to open new opportunities for researchers interested in accurate and fast muscle simulation provided by MuJoCo combined with RL.

8.2 Related work

8.2.1 Building musculoskeletal models

The NeurIPS conference has been hosting recurrent competitions to bridge the gap between biomechanics and reinforcement learning [1] , where the goal was to make a musculoskeletal human model walk. The challenge used the OpenSim (Delp et al., 2007; Seth et al., 2018) simulator. However, as documented in many of the solutions (Pavlov et al., 2018; Zhou et al., 2019; Akimov, 2019; Kolesnikov & Khrulkov, 2020),

[1] https://www.crowdai.org/challenges/nips-2017-learning-to-run
https://www.crowdai.org/challenges/nips-2018-ai-for-prosthetics-challenge
https://www.aicrowd.com/challenges/neurips-2019-learning-to-move-walk-around

the engine was too slow to properly use traditional deep RL methods, which often require millions or billions of samples to find good solutions. Moreover, the model used in the latest challenge was in many ways quite simplistic, actuating only the legs with 22 muscles in total and removing the arms.

Simulating musculoskeletal animal models is not common outside the field of biomechanics. Most of the efforts are focused on humans because of available resources in anatomical atlases and interest from the entertainment industry and sports. Available full-body animal musculoskeletal models include, for example the chimpanzee (Sellers et al., 2013) and dog (Stark et al., 2021). Moreover, for human models, researchers tend to have access to much higher quality motion capture data than for animals, but animal research often have superior empirical measurements of physiological function.

8.2.2 Controlling musculoskeletal models

Some innovative research has used deep RL to solve complex locomotion tasks such as playing basketball (Liu & Hodgins, 2018), performing athletic jumps (Yin et al., 2021), or boxing and fencing (Won et al., 2021). These studies used motion capture data to build a rich reward signal during training (Merel et al., 2017, 2018b), as explored in Chapter 7. Wang et al. (2012) was one of the first studies in the graphics community to use biological actuators. Geijtenbeek et al. (2013) used evolution-based algorithms to control muscular bipeds and also optimized muscle routing. Jiang et al. (2019) tried to bridge torque-actuated models and muscle dynamics, using neural networks to map muscle activations to torque commands and achieved natural looking gaits. Lee et al. (2018b) used volumetric muscles and trajectory optimization for juggling. Lee et al. (2019) combined RL and motion capture clips to control a human musculoskeletal model solving a variety of tasks from walking to weight lifting, separating muscle coordination from trajectory mimicking through two different networks with privileged information and an intermediate proportional derivative (PD) controller.

8.3 Methods

This chapter describes how a fast and realistic musculoskeletal model of an ostrich was built by combining several sources of information, including an existing OpenSim model of legs, a dissection, a CT scan, and reinforcement learning tasks that allowed us to iteratively improve properties of the model. This workflow is represented in Figure 8.2.

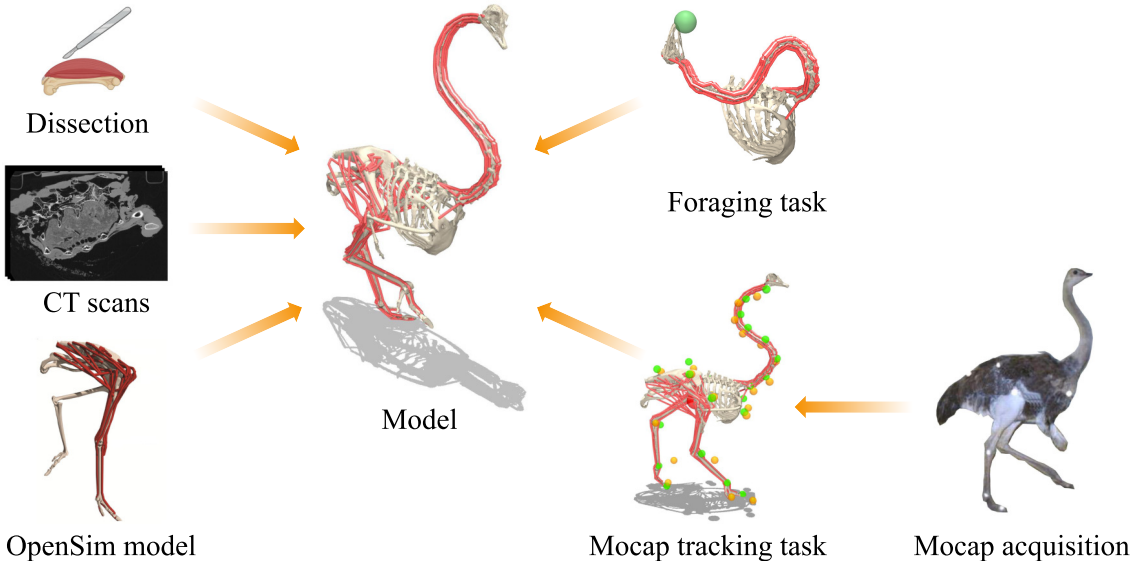


Figure 8.2: The workflow used to build the ostrich model. Various sources of data were blended together for the modeling part, and tasks helped in fine-tuning muscle strength and joint limits.

8.3.1 Anatomical data acquisition

Computed tomography scanning We first acquired multiple computed tomography (CT) scans of a 96.5 kg adult male ostrich specimen, donated by a local farm. The CT scans were then segmented (separating the bone from the rest of the x-ray slices) using the Mimics software (Materialise, Inc.) to obtain 3D models (polygonal meshes) for each bone from the ribcage to the head, including the wings. These 3D geometries were important for two reasons: firstly, they provided an accurate representation of the geometry of the bones, vital for defining muscle attachment points, and secondly, they allowed inference of the inertia of body segments. To obtain the inertia of body segments we followed the steps provided in Hutchinson et al. (2007). The idea is to wrap the skeleton parts within meshes representing flesh and assume constant density. From there, volumes, masses, and the inertia tensors were computed.

Muscle dissection Next, we performed a dissection in order to gather data for muscles of the neck and rib cage. Wings are not actuated in our model because they are mostly used when turning at high velocity and air friction is not considered in our simulations. Dissection is important to properly understand muscle routing, where the muscles start (origin) and where they end (insertion). During the dissection, we used anatomical descriptions from Böhmer et al. (2019), Tsuihiji (2005), and Tsuihiji (2007) to identify the different muscles. We also acquired muscle-specific

data: muscle mass, pennation angle, tendon length, and muscle fiber lengths, used when simulating the muscle-tendon dynamics.

8.3.2 Modeling

OpenSim and MuJoCo Two computer modeling and simulation software packages are commonly used by the biomechanics and reinforcement learning communities respectively. The first one is OpenSim (Delp et al., 2007; Seth et al., 2018), an open source engine that provides benchmarked physics-based muscle simulations. It is relevant to our study not only for its popularity, but also because the legs of our ostrich model were originally modeled with this engine (Hutchinson et al., 2015). The same model was also used to run some tracking (inverse) simulations to estimate the forces, activations, and mechanical work of the muscles involved in solving locomotion tasks (Rankin et al., 2016). OpenSim is rather slow in simulating each time step, as documented in previous NeurIPS competitions that used it to simulate muscle dynamics (Kolesnikov & Khrulkov, 2020). The second simulator, MuJoCo (Todorov et al., 2012), has recently been open-sourced, and is used in multiple RL domains, because it provides fast and accurate rigid body dynamics. Specifically, tendon routing in OpenSim uses an iterative algorithm, while MuJoCo's tendon routing is closed-form and, therefore, much faster. In a comparison made by Ikkala & Hämäläinen (2020), when calculating the average run time over 97 forward simulations, MuJoCo was roughly 600 times faster than OpenSim.

Musculoskeletal models Both OpenSim and MuJoCo define their models in a hierarchical tree structure. Muscles are attached to at least two bones, using two sites at the extremities, called the origin and the insertion sites. Other sites, called waypoints, and wrapping geometries, can also be specified to define elaborate muscle routings. Waypoints are sites through which the muscle must pass, which are useful to maintain an anatomically realistic 3D path for the muscles. Wrapping geometries are useful to prevent muscles from penetrating the bones, and are geometric primitives such as spheres or cylinders that muscles must wrap.

Proposed model After converting the OpenSim leg model to MuJoCo's format, we completed the ostrich model by adding the ribcage, wings, neck, and head geometries. We then actuated the neck by adding the muscles, matching the routing from the dissection. We also modified the morphology of the feet, which were initially flat, to make their shape more realistic. We found that fine tuning the muscle lengths was necessary to improve stability. To find the new muscle ranges, we randomized the model joint angles in their limits a large number of times and recorded the

Figure 8.3: The skeleton of the model follows a tree structure of bodies and joints. The muscle routing is defined using waypoints, shown in green on the left. The joints are shown in light blue in the middle. The wrapping geometries are shown in blue on the right (some wrapping geometries have been omitted for clarity).

global minimum and maximum lengths for each muscle. In our final model, visible in Figure 8.3, each leg contains 7 hinge joints: 3 for the hip, 2 for the knee, 1 for the ankle, and 1 for the metatarsophalangeal (mtp), while the neck contains 36 hinge joints, 2 for each pair of vertebrae and for the head. These 50 joints are actuated by 120 muscles, 68 in the legs and 52 in the neck, making this model very challenging to control. In comparison, the OpenSim model used for the NeurIPS 2019: Learn to Move - Walk Around challenge had only 22 muscles.

Simulation for RL MuJoCo has already been used in a number of popular control domains for reinforcement learning research, in particular dm_control (Tassa et al., 2020) and OpenAI Gym (Brockman et al., 2016). Most of these domains use simplistic models, often inspired by animal morphologies, made up of basic geometric bodies such as capsules, and torque-controlled multiaxial joints. While such models are fast and fairly easy to control, they do not provide a realistic basis for studying animal movement. On the other hand, simulations used in biomechanics typically model animals more accurately (Sellers et al., 2013; Hutchinson et al., 2015; Stark et al., 2021) but are slow and often used in conjunction with relatively simple control techniques from the trajectory optimisation repertoire, such as direct collocation.

8.3.3 Muscle dynamics

Muscle excitation and activation In biomechanics and neuroscience, researchers separate muscle excitation and activation. A neural excitation u, produced by the nervous system, is responsible for the contraction of muscle fibers via an interme-

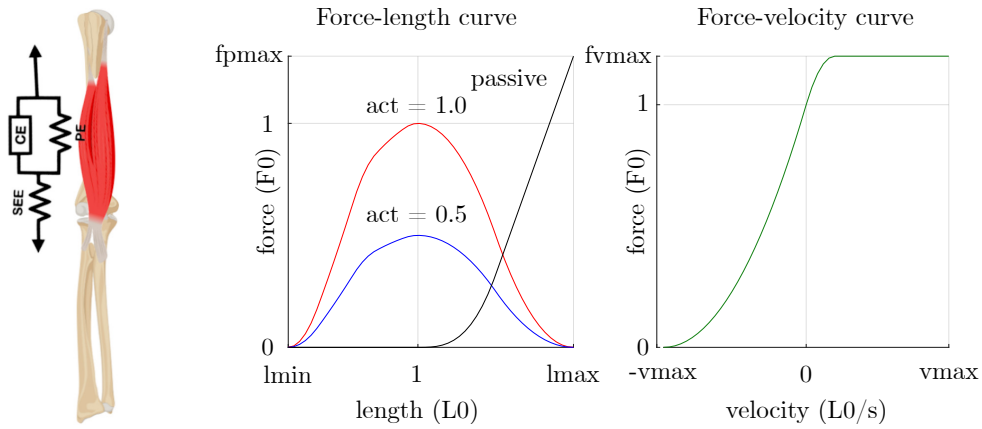


Figure 8.4: On the left a diagram of the Hill-type muscle model comprising the 3 components: a contractile element, a passive element and an elastic element. On the right two plots showing the curves for the force-length and force-velocity functions used in the contraction dynamics.

diate state called activation a (Zajac, 1989). This intermediate state converts an electrochemical signal to mechanical force output. In MuJoCo, this conversion is modelled as a first-order nonlinear filter:

$$\frac{\partial a}{\partial t} = \frac{(u - a)}{\tau(u, a)}$$

where τ is defined as:

$$\tau(u, a) = \begin{cases} \tau_a(0.5 + 1.5a) & u > a \\ \tau_d/(0.5 + 1.5a) & u \leq a \end{cases}$$

τ_a, τ_d are time constants for activation and deactivation, equal by default to 10 ms and 40 ms, respectively.

Contraction dynamics The contraction dynamics are implemented using the Hill-type model as shown in figure Figure 8.4. This mechanical model computes the total force generated by a muscle by adding together the active contractile and passive properties of muscles. The contractile element models the muscle contraction force, scaled by the activation signal a. The passive properties are modeled like springs: if a muscle is stretched, it will generate a passive force to return to its rest position.

The active contraction dynamics can be summarized by the formula:

$$f(l, \dot{l}, a) = a \cdot f_l(l) \cdot f_v(\dot{l}) + f_p(l)$$

where f_l is the active force as a function of muscle length, f_v is the active force as a function of velocity, and f_p is the passive force which is always present regardless

of activation. The force-length and force-velocity curves are showed in Figure 8.4. These functions are built into MuJoCo, capturing the contraction dynamics for basic purposes, and have been validated by the biomechanics community for some conditions (Millard et al., 2013).

All muscle-related quantities in MuJoCo are scaled by the muscle-tendon actuator's resting length. The advantage of this representation is that all muscles behave similarly. MuJoCo does not allow specification of the muscle resting length L_0 and tendon slack length LT directly, in contrast to OpenSim. This is due to the fact that MuJoCo does not include a stateful elastic component in the muscle model. This is a major compromise in MuJoCo when compared to OpenSim, that allows it to run faster. Instead, the actuator length range LR needs to be specified, which is the minimum and maximum of the sum of muscle and tendon lengths, along with a range R in units of L_0. In other words, the actuator length range LR defines the interval of values that the actuator is allowed to use during the simulation and the range R is a percentage expressing how much the actuator (here simply called "muscle") can shorten or extend.

At model compile time, MuJoCo solves these two equations where the unknowns are the muscle rest length L_0 and tendon length LT:

$$(LR_{min} - LT)/L_0 = R_{min}$$
$$(LR_{max} - LT)/L_0 = R_{max}$$

8.3.4 Motion capture data

Original mocap data We did not find any open-source motion capture database of ostrich movements, but after contacting the authors of Jindrich et al. (2007), we obtained the data originally used to study joint kinematics of ostriches performing cutting maneuvers. The provided dataset was composed of 82 clips, recorded at 240 Hz, containing the 3D coordinates of 14 markers distributed as follows: 1 the head, 1 on the spine, and 1 on each breast, hip, knee, ankle, mtp, and toe.

Cleaning of the mocap clips We selected the largest part of the clips where at least 10 markers were simultaneously present for at least 1 second, typically disregarding the beginning and the end of the clips when the ostrich was not in the field of view of the cameras. This selection left 35 clips. However, even in those clips, we found that a large number of markers were periodically missing, probably due to feathers and limbs occluding them from the cameras, as can be seen in Figure 8.5. To overcome this issue, we used a bidirectional LSTM trained in a similar way

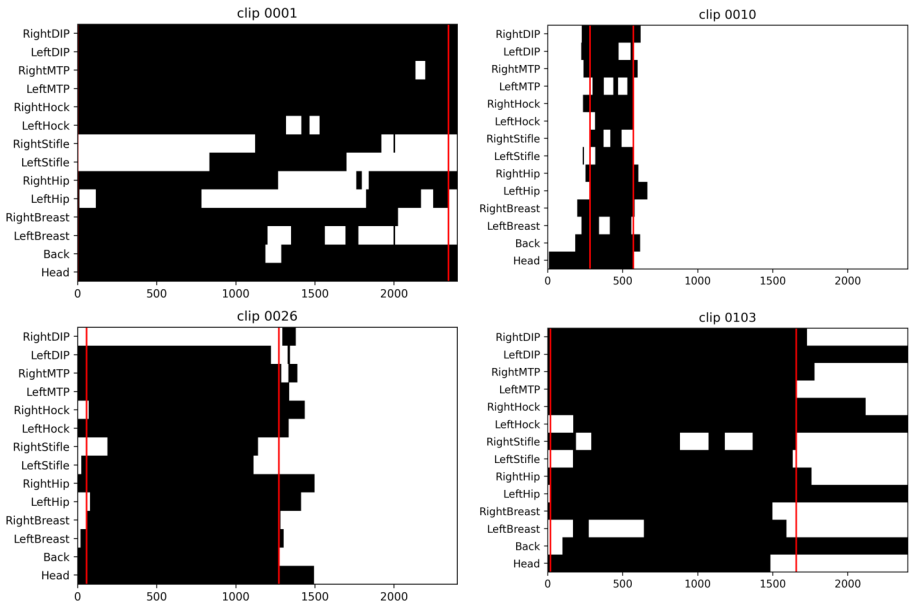


Figure 8.5: Binary masks showing, in black, the available data for the 14 markers, on 4 clips. The red lines indicate the intervals we used. The remaining missing markers had to be predicted.

as a denoising autoencoder, as shown in Figure 8.6. With a probability of 0.1, we randomly masked some markers and asked the model to predict their value given the context of a segment of 100 steps from the same clip. We then used this model to predict the actual missing markers. Since the model was trained and then used on the same, relatively small dataset, we anticipated some overfitting issues, but playing the clips with the predicted markers in place of the missing ones gave surprisingly good results, as shown in Figure 8.7.

Conversion to joint values After rescaling the marker coordinates to match the size of our model, the next step was to transform the set of marker coordinates to joint positions. Fortunately, enough markers were present in the different parts of the ostrich body to limit the space of solutions, except for the neck, where only one marker was present on the head. We used gradient descent with the mean Euclidean distance between the reference markers and those produced by joint poses on two sets of parameters simultaneously. The first set described where to attach the markers on the body parts and is shared across all the clips. This was needed because we did not have the exact location of the markers used during the motion capture acquisition and how they would translate to locations with respect to the bones. The second set of parameters described the joint values at every step. To reduce the space of solutions for the neck shape, we added a regularization term, encouraging the neck to follow an S shape. We first tried to use a finite difference approximation of the

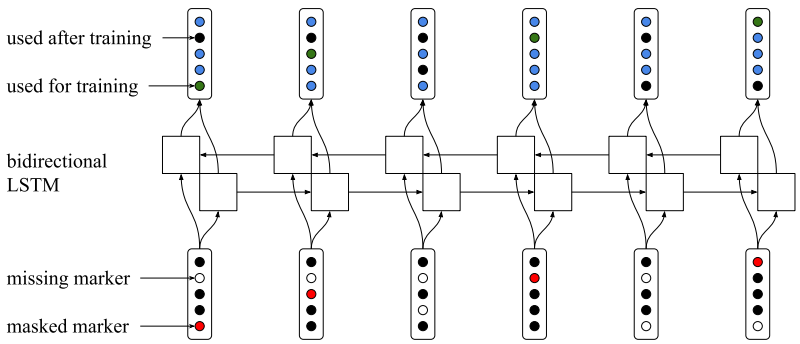


Figure 8.6: Illustration of the bidirectional LSTM used to predict the missing markers. The sequence of masked coordinates are encoded, then a sequence of decoded coordinates are produced, including predictions for the masked ones.

Figure 8.7: Visualization of the mocap prediction process. Segments are drawn between pairs of existing markers. Left: The original incomplete data. Right: The complete data after predicting the missing markers with a bidirectional LSTM.

gradient, but found this approach to be too slow to be practical. We then decided to implement the kinematics function in a differentiable way, mapping joint values to body locations to create marker locations. This function was implemented using differentiable operations from the TensorFlow library (Abadi et al., 2016). We found this approach to be particularly fast thanks to the possibility to simultaneously optimise for all the steps across all the clips using the batch dimension. Finally, the velocity of the joints was inferred using the rate of change between two consecutive steps.

Cyclic clip We also created a cyclic running clip that can be used as a reference for running over an extended period of time. We used the middle section of one of the clips, which we repeated several times, and smoothed to remove discrepancies at the boundaries. This clip helped us produce the simulated biomechanical data for a cyclic gait that we compared to muscle excitations and lengths measured on emus in Subsection 8.4.5.

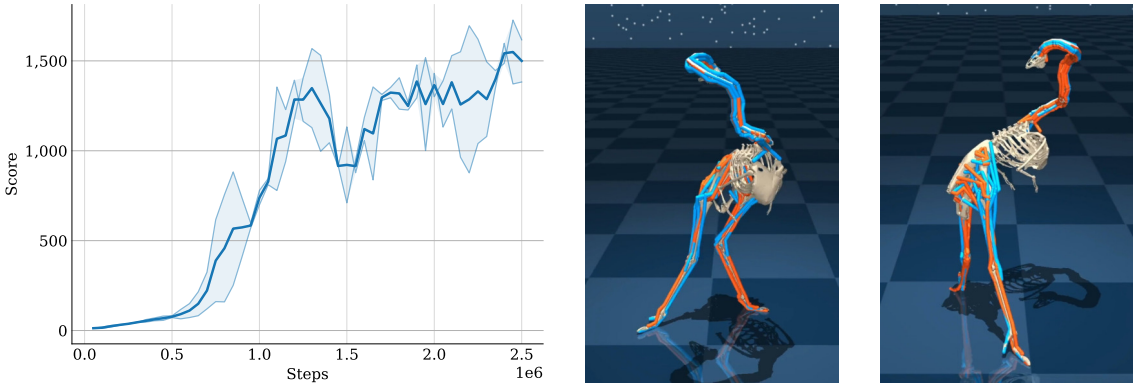


Figure 8.8: Performance on the "run forward" task. Sums of rewards are averaged over 2 seeds and smoothed with a sliding window of size 3. The agent manages to achieve satisfactory performance, but the gaits do not look natural as shown in the videos and images. Muscle excitations from -1 to 1 are represented by colors ranging from blue to orange respectively.

8.4 Experiments

The proposed model was iteratively improved using a set of reinforcement learning tasks. The performance and visual quality of the solutions allowed us to select values for some muscle parameters such as muscle length ranges and forces, or to identify issues with unrealistic joint ranges. We focused on two types of tasks: locomotion and neck control. We found these to be particularly useful when designing the model, because they used the two main groups of muscles present in the model.

8.4.1 Running forward

The first task we experimented with had a simple "move forward" objective, rewarding agents for running as fast as possible. It was initially constrained to a vertical planar space, it used torque-driven joints, and the neck was kept rigidly attached to the model's thorax. The goal was to ensure that the underlying model and simulation structure functioned as intended before adding muscles. The resulting gait was similar to that obtained when training on the 2D walker tasks of dm_control and OpenAI Gym. After adding muscles to the legs, we found that the same gait repeatedly emerged: the toes were unrealistically bent backwards (plantarflexed) and the ankles remained straight. After a series of model adjustments on the different tasks, we were able to achieve more diverse and efficient policies on this one, even after relaxing the model constraint from 2D to full 3D space and actuating the neck. The final "running forward" task is described below, and the results can be seen in Figure 8.8.

Initialization The model is initialized in an upright pose with small random perturbations added to the leg and neck joints. Concretely, for these joints, the initial values are sampled uniformly in 1/5th of their respective intervals, centered around a default pose.

Observations The agent perceives the height of the head, pelvis, and feet, the joint positions except for the x-axis, the joint velocities, the muscle forces, activations, lengths and velocities, and the forward velocity of the center of mass on the x-axis.

Reward and termination The reward is simply the forward velocity of the center of mass on the x-axis. Episodes end if the head height falls below 0.9 meters, if the pelvis height falls below 0.6 meters, or if the torso rotation around the y-axis falls below -0.8 or exceeds 0.8 radians.

8.4.2 Motion capture tracking

The previous "run forward" task showed that learning to control such a complex articulated model with a simple reward function does not sufficiently regularize the search space, producing policies that are both unnatural and suboptimal. A typical solution to better constrain the policy space is to design a more sophisticated reward function. Reference motion tracking is a particularly well-suited option that has been used with motion capture data in many studies to produce natural-looking motions (Liu et al., 2010; Peng et al., 2018; Chentanez et al., 2018; Bergamin et al., 2019; Merel et al., 2018b; Peng et al., 2019), and is also explored in Chapter 7. As a proof of concept, we started with reference motion data that we labeled by hand, frame by frame, using videos of running ostriches found online. The task was initially planar and the model torque-driven. The quality of the results produced encouraged us to continue in this direction.

After producing a satisfactory dataset, described in Subsection 8.3.4, we experimented with a number of variations for the task components. By performing large sweeps over the 35 clips and the cyclic one, we arrived at the mocap tracking task described below. Examples of tracking on two different clips is illustrated in Figure 8.9, the performance on every clip is shown in Figure 8.10 and some of the videos demonstrate the quality of the tracking. The result on short walking clips is very satisfactory. These are the curves that quickly converge. For the most challenging clips, including standing and abrupt changes in speed, learning takes more time, and it seems that longer training would be needed.

Figure 8.9: Examples of motion capture tracking taken from two different clips. Left: A locomotion clip. Right: A complex clip requiring to stand still, perform neck movements, and turn in place. The reference is shown in white next to the model.

Initialization and step During training, an initial time-step is uniformly sampled over the length of the episode, ignoring the last 20 steps. The model is initialized in the pose corresponding to this step, and a small amount of Gaussian noise, with a scale of 0.02, is added to help diversify experiences. The velocity of the joints was set to the one of the reference without modification. Since the control frequency used in our experiments was 40 Hz, the task used every sixth data point of motion capture data (240 Hz) during tracking.

Observations The agent perceives the height of the pelvis and feet, the joint positions and velocities, the muscle forces, activations, lengths and velocities, and the time left in the clip to allow time-dependent policies and to deal with the finite horizon, as described in Chapter 10.

Reward and termination The tracking reward is defined as a product of Gaussian kernels $\exp(-w_p e_p) \times \exp(-w_r e_r)$ across all body parts. The quantity $e_p = \|\bar{p} - p\|$ measures the Euclidean distance between the coordinates of the center of mass p and the reference $\bar{p}$, while the quantity $e_r = \arccos\left((\mathrm{tr}(\bar{R}R^T) - 1)/2\right)$ measures the angle of the difference rotation between the orientation matrix of inertia R and the corresponding reference $\bar{R}$. The weights w_p and w_r control the wideness of the Gaussians, accounting for the different magnitudes and importances. We used the values 0.2 and 0.1 respectively. The reward is bounded in (0, 1] and the multiplicative nature ensures that if one of the components is too far from the reference, the whole reward decays to 0. Episodes end when the reward falls below 0.01 or when the end of the clip is reached.

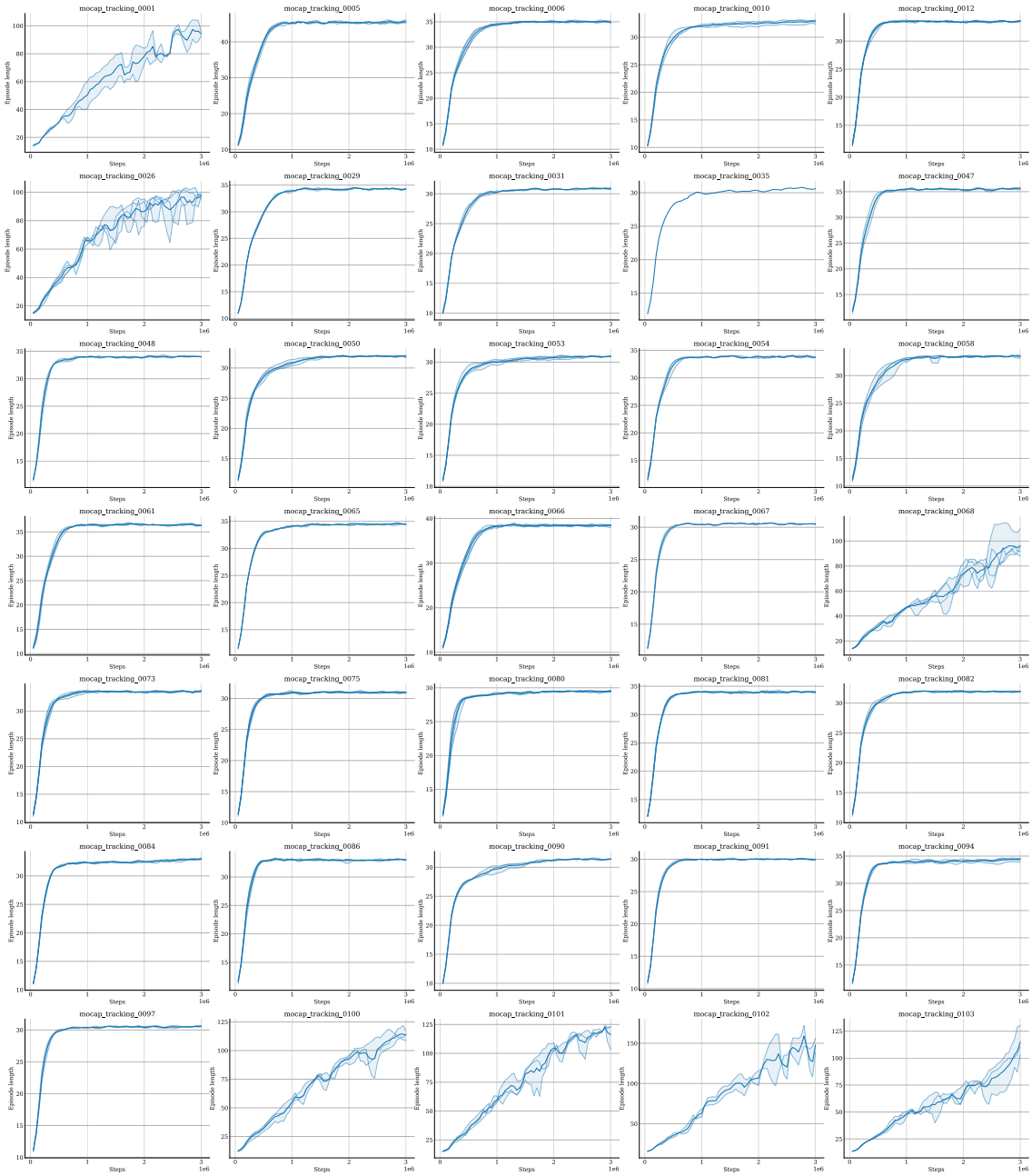


Figure 8.10: Average episode length for all the clips. As agents perform better at tracking, the episodes last longer. Values are averaged over 5 seeds and smoothed with a sliding window of size 3.

8.4.3 Neck control

The neck is composed of 17 vertebrae, which allows for a wide range of shapes. To achieve a particular head position and orientation, an infinite number of solutions exist, and during running, the weight and pose of the neck greatly influence balance. Controlling the neck with muscles that cross several vertebrae therefore requires a sophisticated policy.

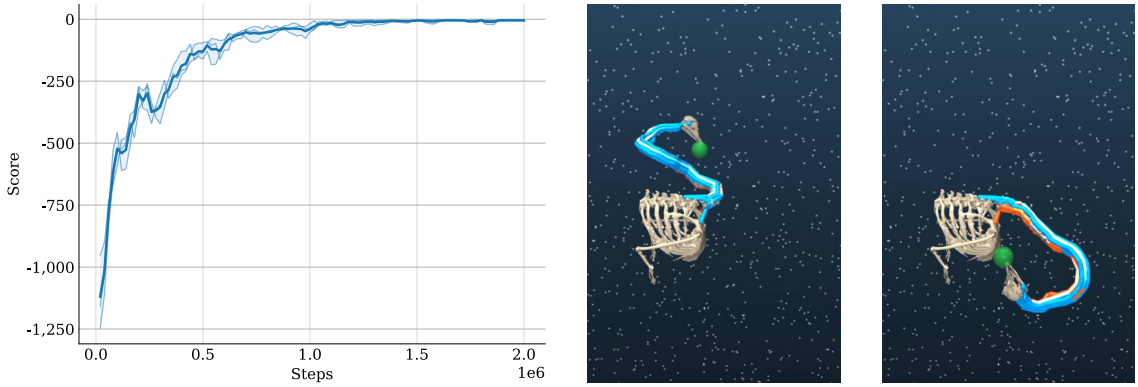


Figure 8.11: Performance on the "neck control" task. Sums of rewards are averaged over 3 seeds and smoothed with a sliding window of size 3. The agent manages to achieve satisfactory performance, and the neck shapes are fairly natural.

To tune the neck portion of the model, we found that mocap tracking of the whole body was not ideal due to the large number of moving parts and the fact that the reference neck poses were artificial. Therefore, we created another task whose objective was to reach random targets with the beak. Only the rib cage, neck, and head were used and the rib cage was held fixed in space. Although effective policies were easily obtained, the initial neck shapes were very irregular, with many abrupt and unnatural changes. To reduce this phenomenon, we increased the stiffness of the neck joints until satisfactory smoothness was achieved. For the joint boundaries, we initially used the values of Dzemski & Christian (2007), but then realized that the boundaries needed to be increased to allow for tighter neck curves that ostriches can perform. The final "neck control" task is described below, and the results can be seen in Figure 8.11.

Initialization Every 100 episodes, and for the very first one, the neck is initialized with a default S shape. For all other episodes, the neck position and velocity are maintained from the last state in the previous episode. The target position is randomized to a feasible area. To do this, we use rejection sampling. Points are repeatedly sampled within a sphere centered at the base of the neck with a radius of 0.8 meters, slightly smaller than the length of the neck, and discarded when inside a second smaller sphere representing approximately the ostrich torso, with a radius of 0.6 meters.

Observations The agent perceives the joint positions and velocities, the muscle forces, activations, lengths and velocities, the coordinates of the beak and target, and the vector from the beak to the target.

Reward and termination The reward is simply the negative of the Euclidean distance between the beak and the target. When this distance falls below a threshold of 0.05 meters, episodes terminate.

8.4.4 Experimental details

To obtain the policies, we used a TD4 agent, proposed in Chapter 12, a mixture of TD3 (Fujimoto et al., 2018), with its pair of critics, delayed actor training and target action noise, and D4PG (Barth-Maron et al., 2018), with its distributional value function (Bellemare et al., 2017) and n-step returns. We also found that Ornstein Uhlenbeck exploration noise (Lillicrap et al., 2015) was significantly better than Gaussian noise, probably due to the importance of correlated excitation when controlling muscles. We chose to use the Tonic deep RL library, described in Chapter 12, because it provided us with a state of the art agent and the flexibility we needed to quickly try, evaluate and visualize experiments while allowing custom dm_control tasks to be used. The different tasks can be visualized on the website.

For the experiments reported above, we used the following hyper parameters. The models are simple MLPs with 2 hidden layers of size 256, ReLU activations, and 51 atoms for the distributional heads. The replay buffer has a size of 1,000,000, batches of size 50 are sampled after the first 50,000 steps and then 50 times every 50 steps. We use 1-step returns, a discount factor of 0.99, a learning rate of 0.0001, a target action noise scale of 0.25, an Ornstein-Uhlenbeck action noise with scale 0.25 and 10,000 initial random steps.

8.4.5 Electromyography comparison

To demonstrate the realism of the proposed simulation, we chose to compare the muscle excitations outputted by a policy trained on the cyclic mocap tracking task, to electromyography (EMG) data. On one side, EMG measures electrical activity in muscles, in response to a stimulation from motoneurons and on the other side, raw actions produced by a policy are used by MuJoCo to produce muscle activations, as explained in Subsection 8.3.3.

Collecting EMG data in animals is challenging because their skin may be too thick for the electrodes and surgery may be required to implant them directly into the muscles. Because no ostrich EMG data have yet been collected, we decided to perform comparisons with data from emus, close relatives of ostriches. EMG data were originally acquired by Cuff et al. (2019) to study muscle recruitment patterns in different bird species and understand neuromuscular control from an evolutionary

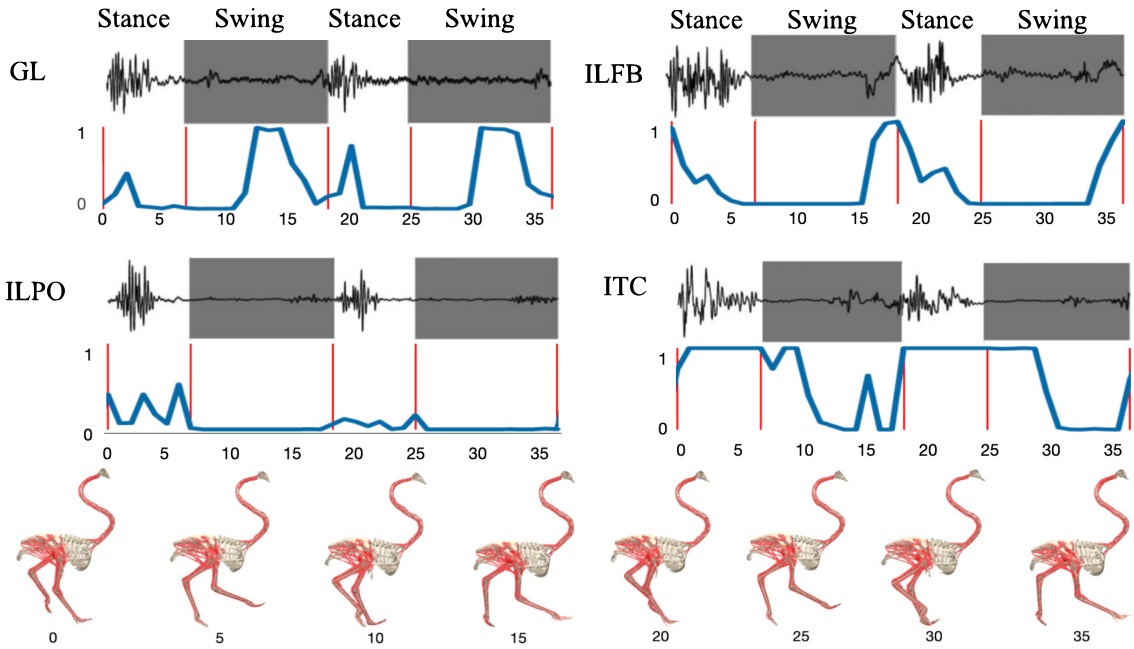


Figure 8.12: Comparison with experimental EMG data from emus. The first red vertical line indicates that the right foot was in contact with the ground (start of stance phase). The second one is when the foot leaves the ground (start of swing phase). EMG data from Cuff et al. (2019) are on top. Muscle excitations (ours) are below.

perspective. Their results showed that walking/running birds generally use their hind limb muscles in very similar ways, making it reasonable to compare emu EMG data with ostrich simulations.

To compare the data, following standard conventions in locomotion research, we divide the gait into two phases: stance (foot is on the ground) and swing (foot is off the ground). Once the gait has been segmented into these phases, we scale them accordingly between the two studies, to account for the difference in speed. As show in Figure 8.12, we find broadly similar excitation patterns, except for the GL (gastrocnemius lateralis) which had an extra burst of swing phase excitation in these simulations, not usually found in any extant birds (Cuff et al., 2019). The ITC (iliotrochantericus caudalis) also had simulated excitation throughout more of the swing phase than in the EMG data, reminiscent of findings for simulated ostriches using the OpenSim model in Rankin et al. (2016). There are many more muscles than actual degrees of freedom in the legs, allowing many possible excitation patterns to match the kinematic data. In addition, because unnecessary excitations are not penalized, it is expected to see more excitations than in the experimental data.

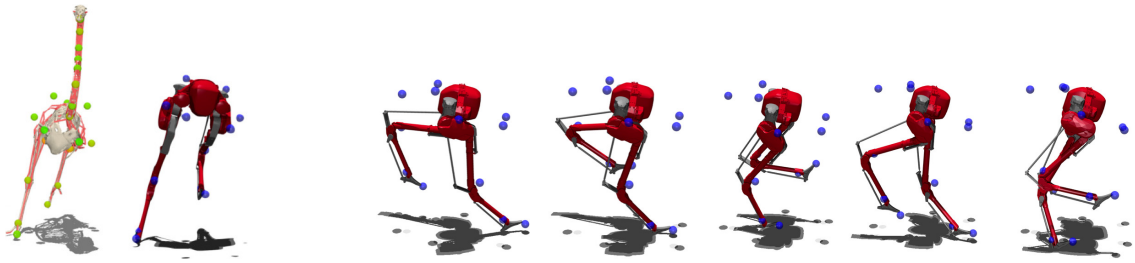


Figure 8.13: Ostrich and Cassie markers comparison, and poses from a motion capture clip. The morphology of the two models is similar enough to allow the motion capture data to be reused.

8.4.6 Compatibility with Cassie

Cassie is a bipedal robot produced by Agility Robotics. It has undergone extensive work in trajectory optimization (Reher et al., 2019; Li et al., 2020; Apgar et al., 2018) and reinforcement learning (Xie et al., 2018, 2020; Li et al., 2021). Its morphology is similar to that of ostrich legs with short thighs, however, because Cassie's joints are powered by electric motors, they are each limited to one degree of freedom. To create multiple degrees of freedom, several joints are added in series and an offset is necessary between them for mechanical reasons. This is a major difference between the ball-and-socket joints allowed by musculoskeletal bodies and traditional robots. In addition, Cassie couples the knee and ankle joints using rods. However, the similarity to the ostrich morphology seemed sufficient to attempt to apply some of our tasks to a model of this robot. This provides an interesting opportunity to compare muscle and torque actuation with two different real-world body models with fairly similar morphologies.

We started with Cassie's MuJoCo model, provided by the Oregon State University Dynamic Robotics Laboratory [2]. The model had to be adjusted to increase its stability and some equality constraints had to be added to ensure that the parts were properly connected with the rods. In order to obtain the robot poses corresponding to the mocap clips, except for the neck, we used the mocap generation pipeline described in Subsection 8.3.4. The only modification was the use of MuJoCo's Jacobian function during the gradient descent step (Buss, 2004), to satisfy the equality constraints. Surprisingly, Cassie's morphology is close enough to that of an ostrich that the clips can be reproduced fairly accurately, as shown in Figure 8.13. The resulting dataset provides an interesting set of behaviors achievable with the Cassie robot, including steps that are more natural than those typically found in the literature and demos using this robot.

[2] https://github.com/osudrl/cassie-mujoco-sim

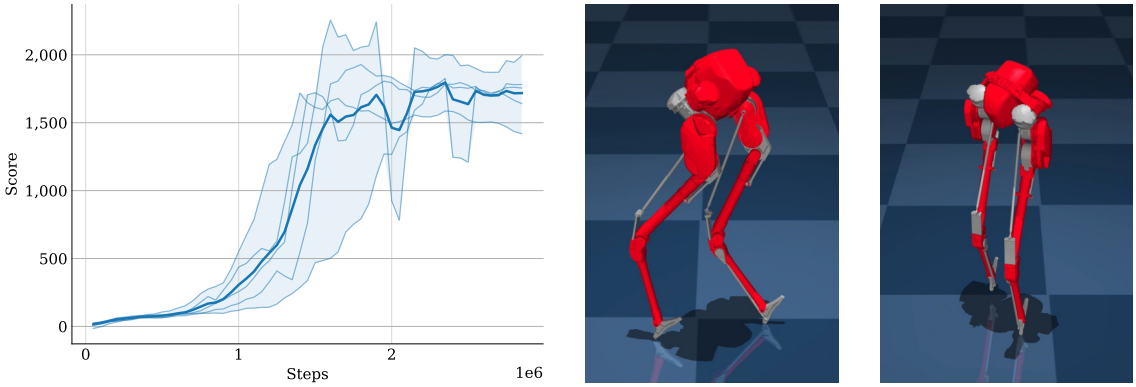


Figure 8.14: Performance on the "run forward" task with the Cassie model. Sums of rewards are averaged over 5 seeds and smoothed with a sliding window of size 3. The agent manages to achieve a slightly faster speed than with the musculoskeletal ostrich model.

Unfortunately, Cassie's model remains quite unstable on the motion capture tracking task, probably due to constraints in the rods and more work would be needed to achieve a satisfactory level of tracking on all clips. However, we also adapted the "run forward" task with more success, as shown in Figure 8.14.

8.5 Discussion

We presented a new musculoskeletal model of an ostrich. This is the only existing complete model of the fastest biped on Earth, an appropriate choice for studying fast locomotion. The proposed MuJoCo simulation is significantly faster than most available musculoskeletal models, usually implemented with OpenSim. Anatomical data were used to ensure that the model was reasonably realistic. We also released a motion capture dataset of ostrich behaviors and open-sourced a set of reinforcement learning tasks built with dm_control. The neck control task is novel and specific to the unique morphology and control aspects of the model. The motion capture tracking task, while similar to those found in the literature, has a unique tracking objective based on matching body part locations and rotations instead of relying on joints and markers. We found that this method produced better tracking results.

The unique combination of a fast and realistic musculoskeletal model with reinforcement learning tasks provides an interesting new benchmark for RL agents in general. Muscle-actuated systems are very different from torque-actuated systems and present specific challenges that we believe could be of interest to the RL community. These include over-actuation, filtered temporal dynamics, and nonlinear force curves. In addition, it is well known that when learning with a simple reward function to control complex articulated models, policies tend to produce odd and suboptimal

behavior. In effect, we found that, compared to motion capture tracking, the running task produced very unnatural and inefficient policies. A major challenge for the RL community could therefore be to try to improve the policies discovered on simplistic tasks. Research directions could involve more efficient exploration methods, new inductive biases in neural architectures, or other types of regularizations, including energy efficiency and fatigue (Potvin & Fuglevand, 2017). We believe that our model and task set are ideal for this type of research. In addition, new tasks can easily be created to study different problems, such as jumping or adapting to rough terrain, and we would be happy to support their inclusion in the repository.

Furthermore, this work is an interdisciplinary effort, with the goal of bridging several research communities. We believe that the interaction between fields such as biomechanics, reinforcement learning, graphics, robotics, and computational neuroscience can produce new understanding of biological bodies and advances in artificial ones. The use of reinforcement learning to improve our model can be more widely accepted for biomechanics research in place of more traditional techniques and can help produce synthetic muscle activation data. This data can in turn help fields such as sports science to create more effective training or recovery programs (Coste et al., 2017). In addition, the computer graphics community is striving for more true-to-life animations, and musculoskeletal simulations could be an important asset in producing more natural movements while introducing more accurate physical interactions and skin deformations when using volumetric muscles. Roboticists have attempted to implement robots powered by artificial muscles, and our work could be used to perform comparisons in energy efficiency and joint movement capabilities. Finally, in computational neuroscience research, models of motor neurons, motor primitives, and muscle synergies could be tested using our simulation.

Part III

Designing infrastructures

Chapter 9

Training infrastructure

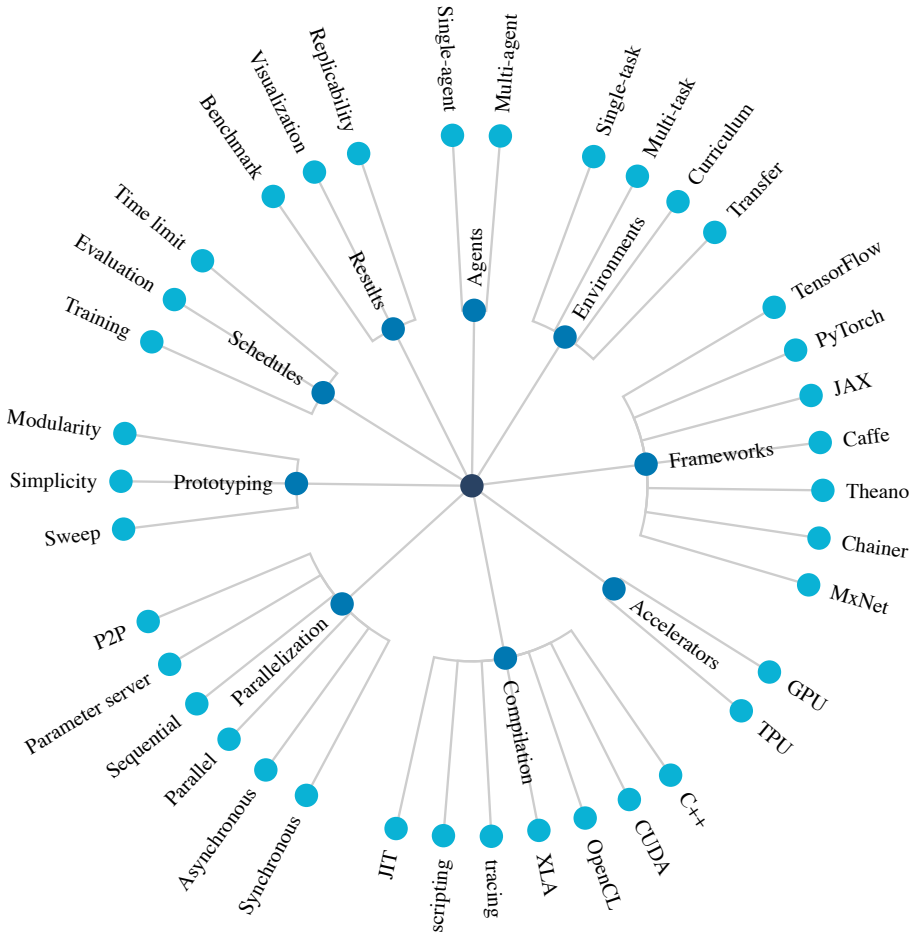


Figure 9.1: Radial layout enumerating a number of elements involved in the design of training infrastructures. These include agents, environments, deep learning frameworks, accelerators, compilation techniques, distributed training, prototyping, training schedules, and results interpretation.

9.1 The infrastructure

While research with supervised and unsupervised learning setups typically involves a rather simple training loop with a dataset and a model, setting up a full reinforcement learning experiment is quite challenging. Various components must be considered to build training infrastructures. Some of the main groups are shown in Figure 9.1 and are described in the next sections. First, at the heart of the infrastructure, the interaction between agents and environments must be handled with care. In particular, communication between the two components and resets can be particularly problematic in the context of distributed configurations as environment resets can happen asynchronously. To optimize the learning speed in terms of wall clock time or number of environment steps, a subtle balance has to be found between the throughput of data coming from interacting with environments and the amount of data used for training simultaneously. In addition to training, evaluations are desirable both in terms of quantitative measures, such as episode scores, and qualitative signals through video recordings and reload of trained agents in viewers.

Deep reinforcement learning libraries A number of libraries have been designed to facilitate experimentation, some dedicated to particular deep learning frameworks or specific types of environments. The rllab (Duan et al., 2016) and OpenAI Baselines (Dhariwal et al., 2017) libraries were arguably the first two widely used collections of deep reinforcement learning agents and provided a very useful starting point for research in this area. Several libraries have since been released and popularized, including Dopamine (Castro et al., 2018), TensorFlow Agents (Guadarrama et al., 2018), Spinning Up (Achiam, 2018), and Acme (Hoffman et al., 2020). Interestingly, there seems to be less consensus for a high-quality, widely used, deep reinforcement learning library compared to deep learning libraries such as TensorFlow, PyTorch, and JAX. This can probably be explained by the fact that while deep learning libraries provide general low-level tools, deep RL libraries contain higher-level algorithms that tend to be less general and subject to preference.

9.2 Infrastructure components

Agents Having a collection of agents readily available is always useful when conducting research on deep reinforcement learning. On the one hand, for work proposing new agents or agent components, relevant baselines are needed to evaluate the benefits of the proposals. On the other hand, for research focused on environ-

ments or any other aspect, a number of robust and battle-tested agents are equally important. It is also worth mentioning that, although this thesis only deals with single agent research, an entire research area is devoted to multi-agent reinforcement learning, in which several agents interact with an environment simultaneously.

Environments Similarly, having a collection of environments is essential. On the one hand, when a new agent is proposed, it must be evaluated on several environments to better appreciate the qualities and drawbacks of the proposed approach. On the other hand, some experiments may involve several tasks at once. For example, research on continual learning focuses on the ability of agents to solve a sequence of tasks while transferring knowledge to new tasks and without forgetting the previous ones. Alternatively, a curriculum of tasks can be used to facilitate the acquisition of skills needed to solve a more complex one.

Deep learning frameworks The implementation of deep reinforcement learning agents relies heavily on deep learning frameworks. Originally, Theano, Caffe, and Torch were very popular in the deep learning field, but this interest has now shifted towards TensorFlow, PyTorch, and JAX. This evolution is largely due to the considerable efforts of engineers at Google and Meta, and the large amount of available compatible code bases developed by the community. While the core ideas that make up an agent should in principle be agnostic, the choice of framework is very important in practice, as researchers may ignore a codebase altogether if the framework used is not the one they favor.

Accelerators Dedicated accelerators play a very important role in large-scale deep learning experiments. Graphics processing units (GPUs) have been developed to accelerate graphical operations, such as rendering in video games. Intensive work is indeed needed for texture mapping, polygons rendering, geometric calculations including rotation and translation of vertices, and programmable shaders. However, their specialization for matrix and vector operations and their massive parallelization capabilities due to hundreds of cores present in their chips naturally made them suitable to accelerate deep learning operations. More recently, tensor processing units (TPUs) (Google, 2018) were developed specifically for deep learning computation with custom code instructions, efficient communication between cores, and high-bandwidth data transfer.

Compilation techniques Most modern neural network implementations are based on the Python language and deep learning libraries. Python is an interpreted

language that offers a very simple interface for developers, but that does not produce fast running native code. Therefore, most deep learning libraries are built with a more efficient language, generally C++, and provide a Python interface to the optimized function calls. However, using fast underlying functions is not sufficient to ensure that inference and backpropagation are fast if the computation graph has to be reinterpreted with every call. Therefore, most deep learning libraries offer mechanisms, often called "tracing", "scripting", or "just in time compilation", allowing graphs to be constructed once and then reused. These optimization steps can for example transform conditional statements and loops into efficient implementations inside of the graph. Furthermore, these compiled graphs are often built with low-level optimization libraries such as CUDA, OpenCL, and XLA that efficiently make use of accelerators.

Distributed training The narrative typically used to describe the reinforcement learning framework involves a single agent interacting with a single environment. This seems intuitively natural, as a living being would learn in this manner. However, a training infrastructure involving a simulation or multiple robots need not be constrained by such a limitation. Learning from multiple parallel streams of interactions has several advantages. First, inference in deep learning models benefits more efficiently from batches of inputs than from single inputs, especially with the use of accelerators. Second, these diverse streams of inputs have the potential to generate more diverse data from which learning can benefit. Several techniques have been proposed to distribute the processes involved. For example, learning and interaction can be performed in parallel, network parameters can be stored in a single process or replicated with peer-to-peer gradient communication, and interactions can be performed synchronously or asynchronously in parallel or sequential environments. At the same time, environment resets must be managed, including the corresponding variable number of observations.

Prototyping When developing ideas, speed of implementation and evaluation is crucial. While some libraries have a highly modular approach, allowing the combination of various components, this often comes at the cost of a steep learning curve and lack of clarity. Conversely, other libraries opt for maximum readability and simplicity at the price of rigidity or code repetition. Unfortunately, simplicity and modularity are somewhat antagonistic, and finding the right balance is difficult. Furthermore, to iterate on ideas, sweeps of module configurations and hyperparameter values are often desirable, for which configuration files and command lines usually provide a good solution.

Training schedules Over the course of a training session, several schedules are usually involved. First, a main external loop is run over a fixed number of training steps or episodes. This main loop is usually segmented into epochs of a fixed number of training steps or episodes and each episode itself is usually limited to a maximum number of steps using time limits. At the end of each epoch, evaluation episodes are usually performed and followed by logging of results. Finally, in addition to this, a scheme over tasks can be added, with a transition from one task to another during the learning process.

Results interpretation Interpreting the results of experiments is also of crucial importance. During training and evaluation, values such as losses and scores are commonly recorded in logging files. Loading these values from multiple runs, calculating statistics such as means, medians and variances, smoothing them with a sliding window, and comparing them to other curves is not straightforward. Furthermore, reloading trained policies with the right configuration of modules and environments followed by visualizing interactions is another challenge. Finally, demonstrating the robustness reproducibility of results is important to ensure that claims can be trusted and that new research can be conducted based on them.

9.3 Elements of infrastructure in the thesis

1. In Chapter 4, a novel technique called action branching and an agent called BDQ are proposed. To evaluate them, relevant baselines, standard tasks, and custom ones are used. These evaluations show a correlation between the number of degrees of freedom and the performance of the agents. The selection of the right tasks to evaluate hypotheses is of crucial importance. The experiments were run on several CPU jobs on cloud computing platforms.

2. In Chapter 5, a novel technique called Q-map, an exploration method based on random goals, and an agent are proposed. The last experiments involving Super Mario All-Stars include runs where exploration alone was tested, others where complete agents were compared, and a proof of concept of Q-map transfer. The infrastructure to enable or disable parts of the agents and record videos of the best episodes was a challenge. The experiments were run on a single desktop computer with two GPUs.

3. In Chapter 7, a novel training pipeline is proposed for learning probabilistic neural motor primitives (NPMPs). The original approach involved independent training of hundreds of policies, recording of a dataset, and distillation of skills

through supervised learning. The proposed approach uses a fully reinforcement learning based pipeline where motion capture tracking and complementary tasks are combined. This work also addresses transferability, as the goal of the low-level controller is to be reused to solve other challenging tasks. The experiments were run on a powerful cluster with hundreds of parallel workers on CPUs and learning performed on TPUs.

4. In Chapter 8, a new musculoskeletal model and reinforcement learning tasks are proposed. Modeling required fine-tuning of muscle lengths and forces, and the workflow used to improve the model involved training on the tasks, evaluating quantitative and qualitative performance, and using these observations to refine the models. This rapid iteration on configuration, training, and interpretation of results was made possible by the Tonic library described in Chapter 12. Experiments were run on a combination of personal computers and cloud computing, both on CPUs and GPUs, distributed when necessary to quickly experiment with new ideas and undistributed when large-scale sweeps and benchmarks were required.

5. In Chapter 10, the distinction between time-limited and time-unlimited tasks is clarified and solutions are proposed to manage both types of tasks. While time limits are often imposed on the environment side, it can be argued that this is more of a training choice, which takes place in the way episodes are scheduled. These considerations were of crucial importance when designing the Tonic library described in Chapter 12. Experiments were run on CPU jobs on cloud servers, with and without distributed training.

6. In Chapter 11, a new deep learning library called Ivy is proposed to unify the most popular deep learning frameworks. In addition, a number of differentiable functions useful for robotics, vision, and control are released with Ivy. When releasing new deep learning code, the choice of the deep learning framework is crucial because a large portion of future users will decide whether to use the proposed work based solely on their familiarity with the chosen framework. Instead, we propose to write libraries using the API proposed by Ivy to maximize cross-framework portability.

7. In Chapter 12, a new deep reinforcement learning library called Tonic is proposed. Its main design principles are organized around facilitating prototyping and benchmarking. In particular, a diverse collection of important agents is proposed, which are assembled using configurable modules written with both TensorFlow 2 and PyTorch. Popular environments are wrapped in a single API, experiment

configuration, trained agent visualization, and result plotting are directly available via command lines, and a collection of baseline results and trained policies are provided. The large-scale benchmarks were run on CPU jobs on cloud computing platforms.

Chapter 10

Learning to act with time limits

This chapter is based on the paper "Time limits in reinforcement learning" (Pardo et al., 2018) published at the ICML 2018 conference. A previous version was first presented at the NIPS 2017 deep RL symposium.

The videos are available at: https://sites.google.com/view/time-limits-in-rl.
The library described in Chapter 12 implements the propositions made in this work.

The main research questions are:

- What are the different types of time limits in reinforcement learning?
- Which methods should be used to handle timeouts in each case?

10.1 Introduction

As explained in Section 2.1, the reinforcement learning framework considers a sequential interaction between an agent and its environment, and the objective of the agent is to maximize its expected return. A return is usually defined as a sum of discounted future rewards, using a discount factor $0 \leq \gamma < 1$ to ensure bounded values.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \tag{10.1}$$

While the series is infinite, it is common to use this expression even in the case of possible terminations, such as timeouts, by considering them to be the entering of an absorbing state that transitions only to itself and generates zero rewards thereafter. However, when the maximum length of an episode is fixed, it is easier to rewrite the expression above by explicitly including the time limit T.

$$R_{t:T} = r_{t+1} + \gamma r_{t+2} + ... + \gamma^{T-t-1} r_T = \sum_{k=1}^{T-t} \gamma^{k-1} r_{t+k} \tag{10.2}$$

Optimizing for the expectation of the return specified in Equation 10.2 is suitable for naturally *time-limited tasks* where the agent has to maximize its expected return only over a fixed episode length. In this case, since the return is bounded, a discount factor of $\gamma = 1$ can be used. However, in practice it is still common to keep γ smaller than 1 in order to give more priority to short-term rewards. The objective of the agent does not go beyond the time limit. Therefore, an agent optimizing for it could learn to take more risky actions leading to higher expected rewards as approaching the time limit. In Subsection 10.3.1, we study this case and explain that due to the time limit terminations, the remaining time is an inherent part of the environment's state and is essential to its Markov property. Therefore, we argue for the inclusion of a notion of the remaining time in the agent's input, an approach that we refer to as *time-awareness* (TA). We describe various scenarios where lacking a notion of the remaining time can lead to suboptimal policies and instability, and demonstrate significant performance improvements for agents with time-awareness.

On the other hand, optimizing for the expectation of the return specified by Equation 10.1 is relevant for *time-unlimited tasks* where the interaction is not limited in time by nature. In this case, the agent has to maximize its expected return over an indefinite period, possibly infinite. However, it can be desirable to still use time limits in order to diversify the agent's experience. For example, starting from highly diverse states can avoid converging to suboptimal policies that are limited to a fraction of the state space. In Subsection 10.3.2, we show that in order to learn good policies that continue beyond the time limit, it is important to differentiate between the terminations that are due to time limits and those from the environment. Specifically, for bootstrapping methods, we argue to bootstrap at states where termination is due to time limits, or more generally any other causes than the environmental ones. We refer to this approach as *partial-episode bootstrapping* (PEB), and demonstrate that it can significantly improve the performance of agents.

Although the importance of accounting for time when optimizing a time-limited objective (finite horizon) is well established in the dynamic programming and optimal control literature (Bertsekas & Tsitsiklis, 1996; Bertsekas, 2012), for example through the use of model-based backward induction, we observed that it has been largely overlooked in the reinforcement learning literature and in the design of popular environment suites. The main contributions of this work are: the analysis of the specific issues which can be caused by the lack of time-awareness, the formalization of partial-episode bootstrapping, and the empirical evaluations demonstrating the relevance of these methods.

10.2 Related work

10.2.1 Learning with a notion of time

The importance of the inclusion of a notion of time in time-limited problems was first demonstrated in the reinforcement learning literature by Harada (1997), yet seems to have been largely overlooked. A major difference between the approach of Harada (1997) (i.e. the Q_T-learning algorithm) and that described in this work, however, is that we consider a more general class of time-dependent MDPs where the reward distribution and transitions can also be time-dependent, preventing the possibility to consider multiple time instances at once. Furthermore, many methods exist to handle partially observable Markov decision processes (Lovejoy, 1991). It is highly common to use a stack of previous observations or recurrent neural networks to address partial observations (Wierstra et al., 2010). These solutions can, to some extent, be useful when a notion of the remaining time is not included as part of the agent's input.

10.2.2 Learning with non-terminal episode boundaries

Related to the proposed partial-episode bootstrapping, White (2017) introduces a way to consider episodic tasks as a continuing one with a variable discount factor between them. Our approach however differs in several ways: (1) PEB is more suitable for tasks that do not necessarily have an underlying episodic structure such as Hopper-v1. (2) In the proposed PEB approach, the agent does not experience a transition from the last state of a partial episode to the first state of the next episode, thus enabling environmental resets based on time limits during training. (3) PEB uses a constant discount factor, allowing the learning of correct indefinite-horizon policies from partial episodes.

10.3 Methods

10.3.1 Time-awareness for time-limited tasks

In tasks that are time-limited by nature, the learning objective is to optimize the expectation of the return $R_{t:T}$ from Equation 10.2. Interactions are systematically terminated at a predetermined time step T if no environmental termination occurs earlier. A time-wise termination can be seen as transitioning to a terminal state whenever the time limit is reached. The states of the underlying MDP thus contain a notion of the remaining time used by its transition function. This time-dependent MDP can be thought of as a stack of T time-independent ones followed by one

that only transitions to a terminal state. Thus, at each time step $t \in \{0, ..., T-1\}$, actions result in transitioning to a next state in the next MDP in the stack.

In effect, a time-unaware agent has to act in a POMDP where states that only differ by their remaining time appear identical. This phenomenon is a form of *state-aliasing* (Whitehead & Ballard, 1991) that is known to lead to suboptimal policies and instability due to the infeasibility of correct *credit assignment*. In this case, the terminations due to time limits can only be interpreted as part of the environment's stochasticity where the time-unaware agent perceives a chance of transitioning to a terminal state from any given state. In fact, this perceived stochasticity depends on the agent's current behavioral policy. For example, an agent could choose to stay in a fixed initial state during the entire course of an episode and perceive the probability of termination from that state to be $1/T$, whereas it could choose to always move away from it in which case this probability would be perceived as zero.

The time-awareness discussed in this work relies on simply including the remaining time to the agent's input. The state-value function at time t for a time-aware agent in an environment with time limit T is therefore:

$$V^{\pi}(s, T-t) = \mathbb{E}_{\pi}\left[R_{t:T} \mid s_t = s\right]$$

By denoting an estimate of the state-value function by V, the target y for a one-step update, after transitioning to a state s' and receiving a reward r, is:

$$y = \begin{cases} r & \text{at all terminations} \\ r + \gamma V(s', T-t-1) & \text{otherwise} \end{cases} \tag{10.3}$$

A time-unaware agent, deprived of the remaining time information, would learn value functions with or without bootstrapping from the estimated value of s' depending on whether the time limit is reached. These conflicting updates for estimating the value of the same state result in an inaccurate average. It is worth noting that, for time-aware agents, the observed time left $T-t$ could be replaced by the elapsed time t. In both cases, the agent could be trained to deal with different episode lengths. However, in the case of the elapsed time observation, a second value indicating the episode time limit should be observed too, or an offset must be added to the observed elapsed time so that the value at the time limit remains the same. In practice, for agents using neural networks, we used a scalar representing the time left, normalized from -1 to 1.

10.3.2 Partial-episode bootstrapping for time-unlimited tasks

In tasks that are not time-limited by nature, the learning objective is to optimize the expectation of the return R_t from Equation 10.1. While the agent has to maximize its expected return over an indefinite (possibly infinite) period, it is desirable to still use time limits in order to frequently reset the environment and increase the diversity of the agent's experiences. A common mistake, however, is to then consider the terminations due to such time limits as environmental ones. This is equivalent to optimizing for returns $R_{t:T}$ from Equation 10.2, not accounting for the possible future rewards that could have been experienced if no time limits were used.

In the case of bootstrapping methods, we argue for continuing to bootstrap at states where termination is due to the time limit. The state-value function of a policy at time t can be rewritten in terms of the time-limited return $R_{t:T}$ and the value from the last state:

$$V^\pi(s) = \mathbb{E}_\pi \left[R_{t:T} + \gamma^{T-t} V^\pi(s_T) \mid s_t = s \right]$$

By denoting an estimate of the state-value function by V, the target y for a one-step update, after transitioning to a state s' and receiving a reward r, is:

$$y = \begin{cases} r & \text{at environmental terminations} \\ r + \gamma V(s') & \text{otherwise (including timeouts)} \end{cases} \tag{10.4}$$

An agent without partial-episode bootstrapping would not bootstrap at timeout terminations. Similarly to Subsection 10.3.1, the conflicting updates to estimate the value of the same state lead to incorrect predictions. While in time-limited tasks, the lack of time awareness leads to bootstrapping from states that are out of reach, in time-unlimited tasks, failure to bootstrap at the end of the time limit obfuscates the fact that more rewards should be available afterward.

10.4 Experiments

10.4.1 Removing state aliasing on LastMoment

To give a simple example of the learning of an optimal time-dependent policy, we propose an environment called LastMoment illustrated in Figure 10.1. The underlying MDP contains two states A and B. The agent always starts in state A and has the possibility to choose an action to "stay" in place with no rewards or a "jump" action that transitions it to state B with a reward of 1. However, state B is a trap with no exit where the only possible action provides a penalty of -1. The

Figure 10.1: Left: Illustration of the proposed LastMoment problem. An optimal policy should wait for the last time step to jump and collect the reward. Right: Illustration of the proposed QueueOfCars problem. An optimal policy should decide to take more risky actions if reaching the exit will take too long.

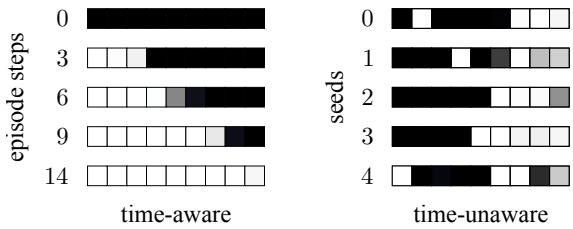
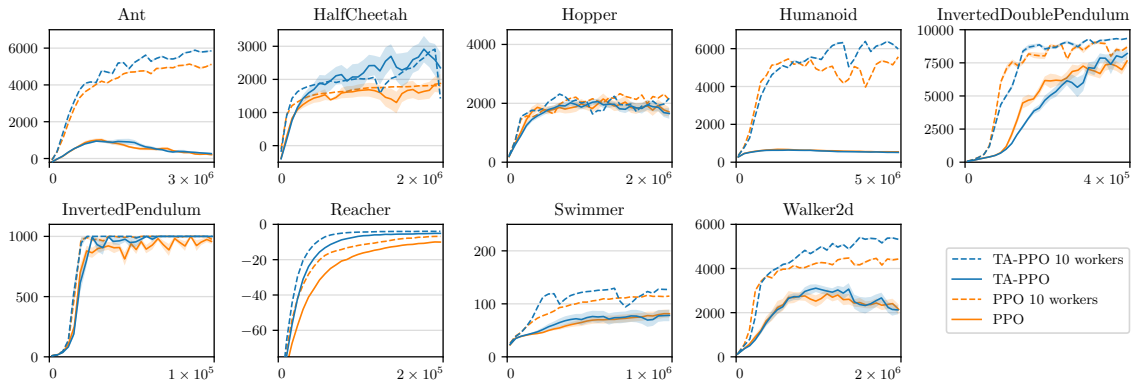


Figure 10.2: Heat map of the learned "dangerous" action probabilities overlaid on our QueueOfCars problem (black and white denote 0 and 1 respectively). The 9 non-terminal states are represented from left to right. The time-aware PPO agent learns to optimally choose actions with decreasing remaining time, while standard PPO learns various suboptimal strategies depending on the initialization seeds.

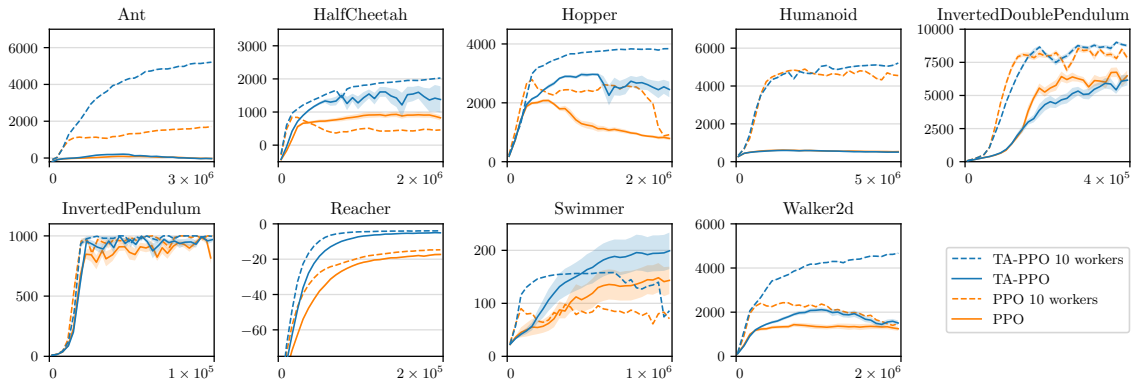
episodes terminate after a fixed number of steps T. The goal of the game is thus to jump just before the timeout. For a time-unaware agent and $T > 1$, the task is impossible to master. The optimal stochastic strategy being to jump 50% of the time when $T = 2$ and to never jump when $T > 2$. In contrast, a time-aware agent can learn to stay in place for $T - 1$ steps and then jump.

10.4.2 Adapting to the remaining time on QueueOfCars

An interesting property of time-aware agents is the ability to dynamically adapt their policy to the remaining time. To illustrate this, we introduce an environment which we call QueueOfCars, pictured in Figure 10.1. The agent controls a vehicle that is held up in a queue of intermittently moving cars. The goal is to reach an exit located 9 slots away from the starting position. At any time, the agent can choose the "safe" action to stay in the queue which may result in advancing to the next slot with 50% probability, or to attempt to overtake with the "dangerous" action which has 80% probability to advance but poses a 10% chance of collision with the oncoming traffic, terminating the episode. The agent only receives a reward of 1 when it reaches the terminal destination. Time-unaware agents cannot possibly adapt to the remaining time and thus can only learn a fixed combination of dangerous and safe actions based on the position. Figure 10.2 shows that a time-aware PPO agent can optimally adapt to the remaining time and its distance to the goal.



(a) $\gamma = 0.99$



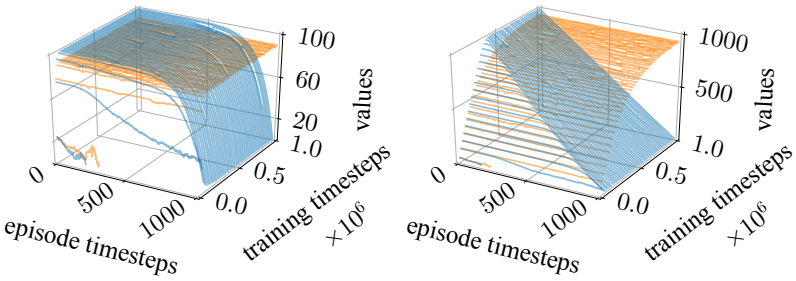
(b) $\gamma = 1$

Figure 10.3: Performance comparison of PPO with and without the remaining time in input on several continuous control tasks from OpenAI Gym ($T = 1000$ for all except $T = 50$ for Reacher-v1). The averaged sum of rewards and standard errors are shown with respect to the number of training steps. On multiple tasks, the proposed time-aware PPO (TA-PPO) outperforms the standard PPO, especially for the case with a large discount factor.

10.4.3 Benchmarking time-aware agents

In this section, we compare the performance of PPO with and without the remaining time as part of the agent's input on 9 continuous control tasks from OpenAI Gym (Brockman et al., 2016; Duan et al., 2016). By default, these environments use predefined time limits which are perceived as environmental terminations. We modified the original TimeLimit wrapper used by these environments to include the remaining time to the state vectors. The results in Figure 10.3 demonstrate that time-awareness (TA) significantly improves the performance of PPO.

As illustrated in Figure 10.3a, for a discount rate of 0.99, often, the standard PPO is initially on par with the time-aware PPO and later starts to plateau (e.g. on Walker2d-v1 and Humanoid-v1). This is due to the fact that, in some tasks, the agents start to experience terminations due to the time limit more frequently



$\gamma = 0.99$ $\gamma = 1$

Figure 10.4: Learned state-value estimations on InvertedPendulum-v1 ($T = 1000$) of the time-aware (blue) and the standard (orange) PPO agents. The first one quickly learns an accurate value while the second one slowly learns an average.

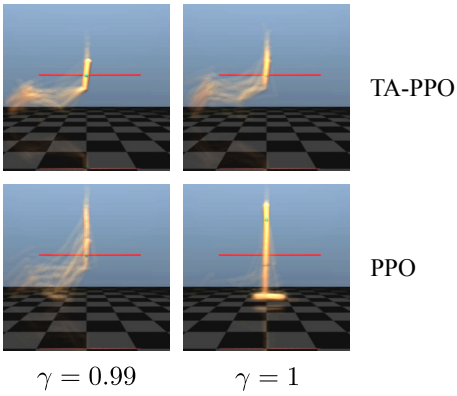
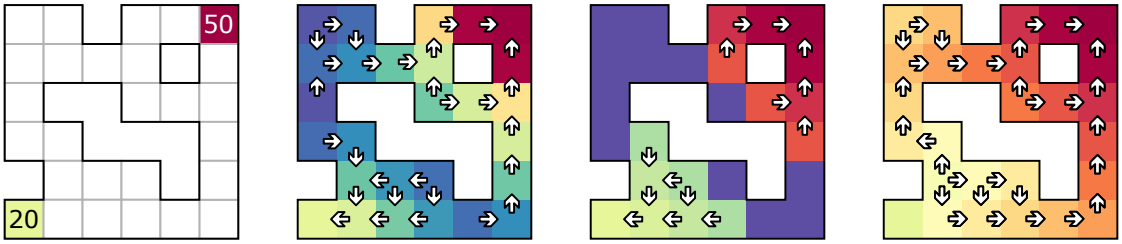


Figure 10.5: Average last pose on Hopper-v1 ($T = 300$) with the vertical termination threshold of 0.7 meters in red. The time-aware agent (TA-PPO) learns to jump forward just before the time limit in order to maximize its forward distance. The time-unaware PPO agent does not learn this behavior and its training is highly destabilized when the discount factor is large.

as they become better, at which point the time-unaware agent begins to perceive inconsistent returns for seemingly similar states. The advantage of time-awareness becomes even clearer in the case of a discount rate of 1 where the time-unaware PPO often diverges drastically as shown in Figure 10.3b. This is mainly because, in this case, the time-unaware agent experiences much more significant conflicts as returns are now the sum of the undiscounted rewards.

Figure 10.4 shows the learned state-value estimations for InvertedPendulum-v1, which perfectly illustrate the difference between a time-aware agent and a time-unaware one in terms of their estimated expected returns. While time-awareness enables PPO to learn an accurate exponential or linear decay of the expected return with time, the time-unaware one only learns a constant average.

Time-awareness does not only help agents by avoiding the conflicting updates. In fact, in naturally time-limited tasks where the agents have to maximize their performance for a limited time, time-aware agents can demonstrate interesting ways of achieving this objective. Figure 10.5 shows the average final pose of the time-aware (top) and time-unaware (bottom) agents. We can see that the time-aware agent robustly learns to jump towards the end of its time in order to maximize its expected



(a) Gridworld (b) Standard (c) Time-awareness (d) PEB

Figure 10.6: Color-coded state-values and policies learned by tabular Q-learning on our TwoGoalsGridworld environment with $T = 3$ and -1 penalties for movements. (a) The gridworld and its two goals. (b) The standard agent perceives timeout terminations as environmental ones and is not time-aware. It always tries to go for the closest goal even if the remaining time is not sufficient. (c) The time-aware agent maximizes its return over the finite horizon and learns to stay in place when there is not enough time to reach a goal. (d) The agent with partial-episode bootstrapping maximizes its return over an indefinite horizon, it learns to go for the most rewarding goal.

return, resulting in a "photo finish". It is also interesting to notice that for $\gamma = 1$, the time-unaware PPO (bottom-right) learns to actively stay in place to at least accumulate the rewards for staying alive.

10.4.4 Handling both time limits on TwoGoalsGridworld

Before switching entirely to time-unlimited tasks, we propose to illustrate the importance of both time-awareness and partial-episode bootstrapping on a single task. We introduce an environment called TwoGoalsGridworld, illustrated in Figure 10.6a. It is a deterministic gridworld with two possible goals rewarding 50 for reaching the top-right and 20 for the bottom-left cells. The agent has 5 actions: to move in cardinal directions or to stay in place. Any movement incurs a penalty of -1 while staying in place generates no reward. Episodes terminate after 3 time steps or if the agent has reached a goal. The initial state is randomly selected for every episode, excluding goals. For every agent, we used tabular Q-learning (Watkins & Dayan, 1992) with random actions, trained until convergence with a decaying learning rate and a discount factor of 0.99.

For the "standard" agent, illustrated in Figure 10.6b, the greedy values of the cells adjacent to the top-right and bottom-left goals converge to 49 and 19, respectively. Then, since $T = 3$, from each remaining cell, the agent has between 1 and 3 steps. If it moves, it receives a penalty and for 2/3 of the times, bootstraps from the successor cell. Thus, for $v(s) = \max_a q(s, a)$ and $N(s)$ denoting the neighbors of s, for states nonadjacent to the goals we have: $v(s) = 2/3(-1 + \gamma \max_{s' \in N(s)} v(s')) + 1/3(-1)$.

This learned value function leads to a policy that always tries to go for the closest goal even if there is not enough time. While the final optimal policy does not actually require time information, this example clearly shows that the confusion during training due to state-aliasing can create a leakage of the values to states that are out of reach. It is worth noting that, Monte Carlo methods such as REINFORCE (Williams, 1992; Sutton et al., 2000) are not susceptible to this leakage as they use complete returns instead of bootstrapping. However, without awareness of the remaining time, Monte Carlo methods would still not be able to learn an optimal policy in many cases, such as the LastMoment problem.

The time-aware agent, illustrated in Figure 10.6c, has a state-action value table for each time step, effectively treating every cell and time step combination as a unique state. The agent properly learns the optimal policy for the time-limited objective, which is to go for the closest goal when there is enough time, and to stay in place otherwise.

In contrast, the agent performing partial-episode bootstrapping, illustrated in Figure 10.6d, always performs bootstrapping unless a goal has been reached. Therefore, if properly learns the optimal policy of always going for the most rewarding goal given a time-unlimited objective and a sufficiently large discount factor.

10.4.5 Connecting partial episodes on Hopper and Walker

In Subsection 10.4.3, we showed that time-aware agents can learn to perform a "photo finish" when trained on locomotion tasks. However, this behavior would lead to a fall and subsequent termination if the interaction was to be extended. Such a policy is not viable if the goal is to learn to move forward for an indefinite period. While one solution could be to not use time limits during training, short episodes remain useful to diversify experiences. In order to use partial-episode-bootstrapping, two key modifications had to be applied to the setup from Subsection 10.4.3. First, we removed Gym's TimeLimit wrapper that is included by default for all environments and which enforces termination when time limits are reached. Second, we modified PPO's implementation to enable bootstrapping when the environment is reset but no termination is encountered. This involves changing the implementation of the generalized advantage estimator (GAE) (Schulman et al., 2015b). Whereas GAE uses an exponentially-weighted average of n-step value estimations for bootstrapping which is more complex than the one-step lookahead bootstrapping explained in Equation 10.4, continuing to bootstrap from the last non-terminal states is the only modification required for the considered approach.

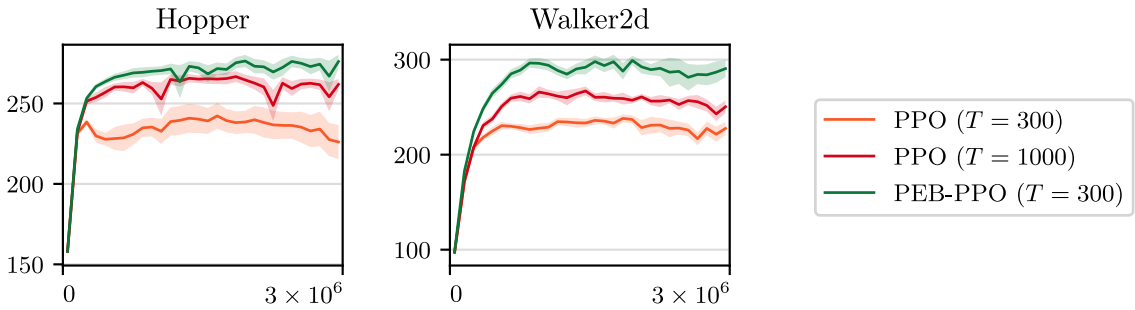


Figure 10.7: Performance comparison of PPO with and without partial-episode bootstrapping with $\gamma = 0.99$ on Hopper-v1 and Walker2d-v1. The averaged scores and standard errors are shown with respect to the number of training steps. The evaluation episodes are limited to 10^6 time steps and the discounted sum of rewards is represented.

Here, we consider the Hopper-v1 and Walker2d-v1 environments and aim to learn a policy that maximizes the agent's expected return over a time-unlimited horizon. The goal is to show that by bootstrapping from states at timeout terminations, it is possible to learn good policies for time-unlimited tasks. Figure 10.7 shows performance evaluations of the standard PPO versus one with partial-episode bootstrapping. During training, partial episodes were limited to 300 time steps, while during evaluation episodes were limited to 10^6 time steps to prevent infinitely long evaluation episodes. The standard PPO agent was also trained with the default time limit ($T = 1000$) for comparison. The results show that partial-episode bootstrapping allows our agent to significantly outperform the standard PPO and that training on short interactions is sufficient.

10.4.6 Connecting partial episodes on InfiniteCubePusher

To demonstrate the ability of our agent to optimize for an infinite-horizon (no terminal state) objective, we propose a novel MuJoCo environment, called InfiniteCubePusher, illustrated in Figure 10.8 (left). The environment consists of a torque-controlled ball, on the horizontal plane, that is used to push a cube to specified target positions. Once the cube has touched the target, the agent is rewarded and the target is moved away from the cube to a new random position. An optimal policy must move the sphere in order to push the cube to the target quickly but in a way that facilitates reaching the following random target too, and because the task lacks terminal states, it can continue indefinitely. The terrain is surrounded by fixed bounding walls. The inner edge of the walls stops the cube but not the ball in order to let the agent move the cube even if it is in a corner. The environment's state representation consists of the coordinates of the ball, the cube, and the target, the

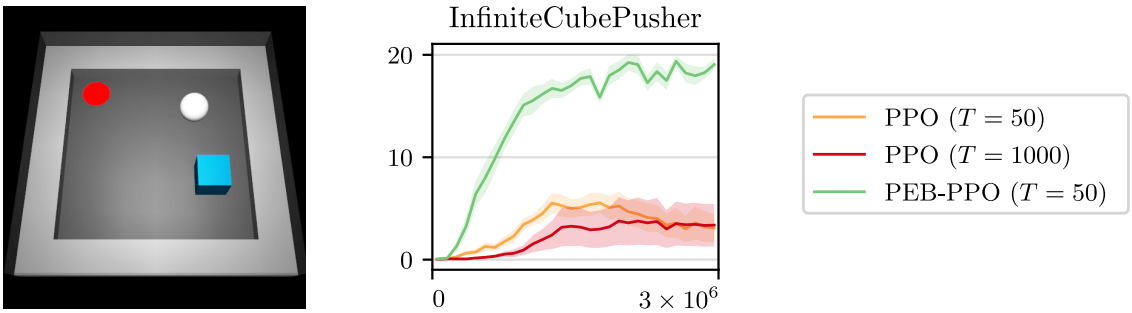


Figure 10.8: Left: Visualization of the proposed InfiniteCubePusher environment. Right: Performance comparison of PPO with and without partial-episode bootstrapping with $\gamma = 0.99$ on InfiniteCubePusher. The averaged scores and standard errors are shown with respect to the number of training steps. The evaluation episodes are limited to 1000 time steps and the number of targets reached per episode is represented.

velocities of the ball and the cube, and the rotation of the cube. The agent receives a reward of 1 every time the cube reaches the target. Due to the absence of *reward shaping* (Ng et al., 1999), it is necessary to limit training episodes in time to diversify the experiences and learn to solve the task. Therefore, during training, a time limit of 50 time steps is used, sufficient to push the cube to one target in most cases. During evaluation, however, 1000 steps were used to allow several targets to be successfully reached. An entropy coefficient of 0.01 was used to encourage exploration. We found this value to yield best performance for both agents. Figure 10.8 shows that our agent drastically outperforms the standard PPO.

10.4.7 Replaying transitions on DifficultGridworld

Sampling batches of transitions from a buffer of past experience, known as experience replay (Lin, 1992b), has proved to be highly effective in stabilizing the training of artificial neural networks by decorrelating updates and avoiding the rapid forgetting of rare experiences (Mnih et al., 2015; Schaul et al., 2015b). However, we argue that the perceived non-stationarity, induced by not properly handling time limits, is incompatible with experience replay. Indeed, the timeout-occurrence distribution changes with the behavior of the agent, and thus past transitions become increasingly obsolete.

While both time-awareness and partial-episode bootstrapping provide ways to solve this issue, we chose to illustrate the effect of PEB on one of the tasks presented in the original manuscript of Zhang & Sutton (2017). In the latter, the authors demonstrate that experience replay can significantly hurt the learning process if the size of the replay buffer is not kept sufficiently small. One of the environments

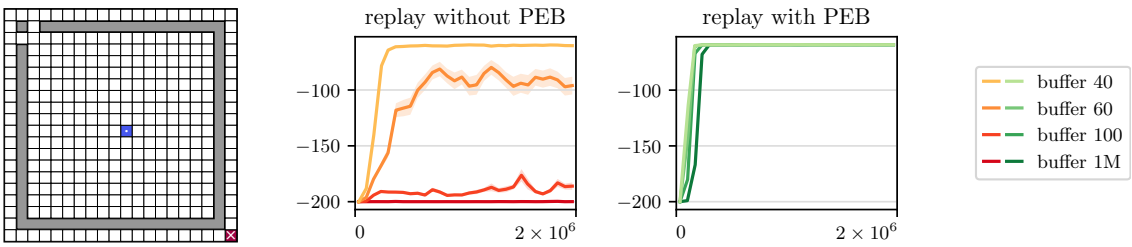


Figure 10.9: Performance comparison of tabular Q-learning with and without partial-episode bootstrapping with $\gamma = 1$ on the DifficultGridworld ($T = 200$) problem presented in Zhang & Sutton (2017). The averaged scores and standard errors are shown with respect to the number of training steps. When timeout terminations are not properly considered, experience replay significantly hurts the performance, while by simply continuing to bootstrap whenever a timeout termination is encountered, the learning is much faster and the buffer size has almost no effect.

used is a deterministic gridworld with a fixed starting state and goal, that we call DifficultGridworld and illustrate in Figure 10.9 (left). As proposed by the authors, tabular Q-learning is used with values initialized to 0, a penalty of -1 at each time step, no discount, a time limit $T = 200$, and an ε-greedy exploration using a fixed 10% chance of random actions. Figure 10.9 shows the performance with respect to the number of training steps, averaged over 30 seeds, from 0 to 29. We successfully replicated the figure showing that the performance deteriorates quickly with buffer size and demonstrate that by simply bootstrapping from states when the time limit is reached, the effect of the buffer size is vastly diminished. It is worth noting that the manuscript from Zhang & Sutton (2017) has since been revised with the proposed PEB method and a reference to our work.

10.4.8 Experimental details

For all experiments involving the PPO agent, we used the OpenAI Baselines (Dhariwal et al., 2017) implementation with the hyperparameters reported by Schulman et al. (2017), unless stated otherwise. The partial-episode bootstrapping version of PPO makes a distinction between environment resets and terminations by using the value of the last state in the evaluation of the advantages if no termination is encountered. For each task involving PPO, we used the same 10 seeds $(0, 1000, ..., 9000)$ to initialize the pseudo-random number generators for the agents and environments.

10.5 Discussion

We have shown that time awareness is necessary for a proper credit assignment with time-limited tasks. However, it should be noted that time-unaware agents

often manage to perform well in the literature. This may be due to several reasons. One of them is that time limits are sometimes very large and therefore almost never experienced, as is the case with the Arcade Learning Environment (ALE) (Bellemare et al., 2013; Machado et al., 2018) where games end after 5 minutes. Some environments might also provide observations that are correlated with time, such as the forward distance, or a distribution of states that is fairly unique as time passes. In addition, a small discount factor may reduce the extent of conflicting updates. However, we argue that providing a notion of time to the agent should be preferred in any time-bound scenario. This method is very simple to implement and can be applied to tasks with varying time limits. It is also interesting to note that knowledge of time can allow agents to learn open-loop (state independent) policies. For example, if a task involves a fixed starting state and no stochastic transitions, then an optimal trajectory can be constructed without using states. Finally, in real-world applications, such as robotics, the wall clock time can be used to allow agents to take into account varying refresh rates and computation times.

We also showed that with time-unlimited tasks, bootstrapping at timeouts was necessary to ensure the learning of optimal time-unlimited policies. However, agents must use reliable value predictions. If timeouts occur in states that are not encountered earlier in episodes, the predictions may not correctly reflect the value of those states. To prevent this from happening, environments must properly randomize the starting states or use sufficiently large time limits to ensure that the state space is properly covered. It is also worth noting that partial-episode bootstrapping is generic in that it is not restricted to early terminations due to time limits but can be applied with any interruption causes other than environmental ones. For example, it is common in curriculum learning to start episodes from states nearby goals and gradually expand to further ones (Florensa et al., 2017b). In this case, it can be helpful to stitch together the learned values by terminating the episodes and bootstrapping as soon as the agent enters a well-known state.

Chapter 11

Building framework-agnostic differentiable functions

This chapter is based on the paper "Ivy: Templated deep learning for inter-framework portability" (Lenton et al., 2021).

The doc is available at: https://lets-unify.ai/ivy.
The source code is available at: https://github.com/unifyai/ivy.

The main research questions are:

- How to unify existing deep learning frameworks to increase code portability?
- How to facilitate development while minimizing overhead?

11.1 Introduction

Software projects usually present a trade-off between efficiency of execution and ease of development. In effect, programming solutions with more abstractions remove complexity, but they also necessarily remove control and the ability to perform task-specific optimizations. Effective frameworks must strike a balance between these two competing factors, where the right abstractions are needed to make development as fast and easy as possible, while allowing custom implementations for maximum runtime efficiency and control.

In the context of deep learning frameworks, Python has emerged as the leading language for research and development, while most frameworks depend on efficient precompiled C++ code. The Python interface makes prototyping code quick and easy, while the background precompiled C++ operations and CUDA kernels accelerate model inference. Although users of most deep learning frameworks still have the option of developing custom operations in C++ and CUDA, the most common use case is for developers to implement their projects as compositions of operations in

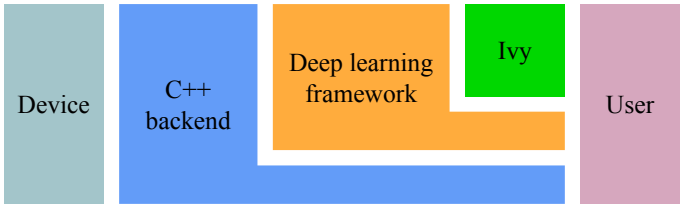


Figure 11.1: Diagram illustrating how Ivy sits on top of and alongside existing deep learning frameworks and C++ backends in the abstraction hierarchy, allowing significant control by the user.

pure Python. The abstractions available for this style of development also continue to grow in power. For example, most frameworks now enable Python functions to be flagged for just-in-time (JIT) compilation, using tools such as the accelerated linear algebra compiler (XLA) (Leary & Wang, 2017).

In this work, we propose Ivy, a new deep learning library that proposes to further simplify code development and reuse by abstracting the choice of a deep learning framework. As shown in Figure 11.1, we frame Ivy in the same abstraction hierarchy as described above. The proposed approach relies on creating a unified functional application programming interface (API) with consistent call signatures, syntax, and input-output behavior. In doing so, Ivy effectively moves existing deep learning frameworks one layer lower in the abstraction stack to the Ivy "backend". As with the abstracted C++ backend in deep learning frameworks. New functions written in Ivy are instantly portable to TensorFlow, PyTorch, JAX, MXNet, and NumPy, enabling an inter-framework "drag-and-drop" approach that is not currently possible among modern deep learning frameworks. If a new Python deep learning framework was introduced in the future, adding this framework to the Ivy backends would then make all existing Ivy code instantly compatible with the new framework. Ivy offers the potential to create "lifelong" framework-agnostic deep learning libraries.

It is worth noting that while deep learning libraries can be used to build trainable parametric modules, they also allow the construction of useful hand-designed parameter-free differentiable functions. In computer vision, for example, multi-view geometry relations such as FlowNet (Dosovitskiy et al., 2015) can be incorporated into computation graphs. Gradient-based optimization can also be used to perform motion planning as demonstrated by works such as CHOMP (Ratliff et al., 2009) and TrajOpt (Schulman et al., 2014). Outside of robotics and computer vision, other fields are increasingly exploiting parameter-free computation in end-to-end graphs. For example Raissi et al. (2020) propose a physics-informed deep-learning framework capable of encoding the Navier-Stokes equations into neural networks with applications in Fluid Mechanics, and Qiao et al. (2020) propose a

differentiable physics framework which uses meshes and exploits the sparsity of contacts for scalable differentiable collision handling. These are just some examples of the growing need for libraries which provide domain specific functions with support for gradient propagation, to enable their incorporation into wider end-to-end pipelines. We provide an initial set of Ivy libraries for mechanics, 3D vision, robotics, and differentiable environments. We expect these initial libraries to be widely useful to researchers in applied deep learning for computer vision and robotics.

11.2 Related work

11.2.1 Deep learning frameworks

Deep learning progress has evolved rapidly over the past decade, and this has spurred companies and developers to strive for framework supremacy. Large matrix and tensor operations underpin all efficient deep learning implementations, and so there is largely more that relates these frameworks than separates them. Many frameworks were designed explicitly for matrix and tensor operations long before the advent of modern deep learning. In the Python language (Van & Drake, 2009), one of the most widely used packages is NumPy (Oliphant, 2006; Harris et al., 2020), which established itself as a standard in scientific computing. NumPy is a general matrix library, but with many function implementations highly optimized in C (Kernighan & Ritchie, 1988). It does not natively support automatic differentiation and back-propagation. Since the beginning of the new deep learning era, a number of libraries with automatic differentiation have been utilized. An early and widely used library was Caffe (Jia et al., 2014), written in C++ (Stroustrup, 2000), enabling static graph compilation and efficient inference. The Microsoft Cognitive Toolkit (CNTK) (Seide & Agarwal, 2016) was also written in C++, and supported directed graphs. Both of these are now deprecated. More recently, Python has become the front-runner language for deep learning interfaces. TensorFlow (Abadi et al., 2016), Theano (Al-Rfou et al., 2016), Chainer (Tokui et al., 2019), MXNet (Chen et al., 2015), PyTorch (Paszke et al., 2019) and JAX (Bradbury et al., 2018) are all examples of deep learning frameworks primarily for Python development.

Despite the variety in frameworks, the set of fundamental tensor operations remains finite and well defined, and this is reflected in the semantic consistency between the core tensor APIs of all modern deep learning libraries, which closely resemble that of NumPy. Ivy abstracts these core tensor APIs, with scope to also abstract future frameworks adhering to the same pattern, offering the potential for inter-framework code reusability long into the future.

11.2.2 Deep learning libraries

Many field-specific libraries exist, for example DLTK (Pawlowski et al., 2017) provides a TensorFlow toolkit for medical image analysis, PyTorch3D (Ravi et al., 2020) implements a library for deep learning with 3D data, PyTorch Geometric (Fey & Lenssen, 2019) provides methods for deep learning on graphs and other irregular structures, and ZhuSuan (Shi et al., 2017) is a TensorFlow library designed for Bayesian deep learning. Officially supported framework extensions are also becoming common, such as GluonCV and GluonNLP (Guo et al., 2020) for MXNet, TensorFlow Graphics (Valentin et al., 2019), Probability (Dillon et al., 2017), and Quantum (Broughton et al., 2020) for TensorFlow, and torchvision and torchtext for PyTorch (Paszke et al., 2019). However, the frameworks these libraries are built for can quickly become obsoleted, also making the libraries obsolete. Furthermore, none of these libraries address the code shareability barrier for researchers working in different frameworks. As of yet, there is no solution for building large framework-agnostic libraries for users of both present and future deep learning Python frameworks. Ivy offers this solution.

11.2.3 Deep learning abstractions

Some previous works do provide framework-level abstractions for deep learning. Keras (Chollet et al., 2015) supported TensorFlow, CNTK, and Theano before its focus shifted to support TensorFlow only. Keras provided abstractions at the level of classes and models, enabling users to prototype quickly with higher level objects and standard training pipelines. However, for researchers adopting non-standard pipelines, this level of abstraction can remove too much control. In contrast, Ivy simplifies and reduces the abstraction to the core tensor APIs, enabling new libraries and layers to be built on top of Ivy's functional API in a highly scalable, maintainable and customized manner.

TensorLy (Kossaifi et al., 2016) is closer in scope to Ivy, also offering a framework agnostic functional API. However, many functions such as gather, scatter and neural network functions such as convolutions are not provided by TensorLy. The library instead focuses on general tensor operations such as decomposition, regression and sparse tensor handling. The TensorLy backend focuses on the functions necessary to support these operations. Ivy offers a much more comprehensive API, applicable to a wide variety of fields, which is showcased in the applied libraries for mechanics, vision, robotics, and differentiable environments.

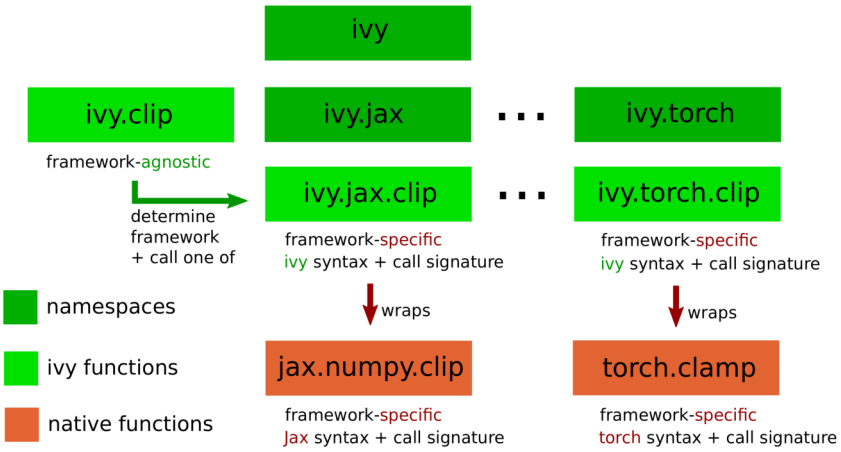


Figure 11.2: Overview of the core Ivy API.

11.3 Methods

We now provide an overview of the core Ivy API and explain how backend frameworks are selected by the user and managed internally.

11.3.1 Core functions

The Ivy Core API provides the low-level building blocks for composing portable Ivy functions and libraries. They are grouped in submodules such as `general`, `math`, `reductions`, `linalg`, `random`, `image` and `logic`. All Ivy functions are unit tested against each backend framework, and support arbitrary batch dimensions of the inputs. The existing core functions are sufficient for implementing a variety of examples through the four Ivy applied libraries, but the core Ivy API can easily be extended to include additional functions. These core functions are essentially wrappers around core functions of the supported frameworks, but some also add new functionalities. For example, `scatter_nd` and `gather_nd` are not present in PyTorch, NumPy and JAX. A few examples are given below.

```
# general
ivy.array(array)
ivy.gather_nd(array, indices)

# math
ivy.sin(array)
ivy.exp(array)

# random
ivy.randint(low, high, shape)
ivy.random_uniform(low, high, shape)
```

Code snippet 11.1: Examples of core functions.

11.3.2 Framework-specific namespaces

The framework-specific functions with Ivy syntax and call signatures are all accessible via framework-specific namespaces such as `ivy.tensorflow` and `ivy.torch`, see Figure 11.2. Each of these namespaces behave like the functional API of the original framework, but with the necessary changes to bring inter-framework unification. Due to the semantic similarity between all deep learning frameworks, these changes are very minor for most functions, with many changes being purely syntactic, relying on direct bindings to the native functions. Other functions require simple rearrangement of the arguments, and sometimes extra processing of optional arguments. We show how Ivy wraps PyTorch functions with varying extents of modification below.

```
# direct binding
clip = torch.clamp

# minimal change
tile = lambda x, reps: x.repeat(reps)

# moderate change
def cast(x, dtype_str):
  dtype_val = torch.__dict__[dtype_str]
  return x.type(dtype_val)

# larger change
def transpose(x, axes=None):
  if axes is None:
    axes = range(len(x.shape) - 1, -1, -1)
  return x.permute(axes)
```

Code snippet 11.2: Examples of binding.

11.3.3 Framework-agnostic namespace

While the framework-specific namespaces provide inter-framework unification of syntax and call signatures, each of these namespaces are still specific to a single framework. In order to create framework-agnostic code, an Ivy namespace is used, with functions accessible directly as `ivy.func_name`. Every function in the framework-specific namespaces is present in the framework-agnostic namespace. They all use `get_framework(*args, f=f)`, which returns the desired framework-specific namespace such as `ivy.tensorflow` or `ivy.torch`, and call the bound framework-specific method.

```
def some_fn(*args, f=None):
  return ivy.get_framework(*args, f=f).some_fn(*args)
```

Code snippet 11.3: Example of framework-agnostic function.

Some Ivy functions are only available in the framework-agnostic namespace, such as `ivy.lstm_update`. These are implemented as compositions of other lower-level functions. With these framework-agnostic functions, new framework-agnostic code can then easily be implemented by composing them in new ways.

11.3.4 Local framework specification

Local framework specification allows the backend to be specified on a per-function basis. In this mode, the framework-agnostic functions are called directly, and the method `get_framework(*args, f=f)` is responsible for selecting the backend. This method determines the desired backend using one of two possible mechanisms.

Framework argument The framework can be specified for any core function call, using the `f` argument, which can either be specified as a string such as `'tensorflow'` or as the backend namespace itself such as `ivy.tensorflow`. When `f` is a string, a dictionary lookup is used, and when `f` is a namespace, `f` is returned unmodified. If `f` is specified, this always takes priority for backend selection.

```
x = ivy.clip(np.array([-1., 2.]), 0, 1, f='tensorflow')
x = ivy.clip(np.array([-1., 2.]), 0, 1, f=ivy.torch)
```

Code snippet 11.4: Examples of framework argument.

Type checking Rather than specifying the framework to use for every function call, the appropriate framework can automatically be inferred by type checking of the inputs. This approach is more user-friendly than specifying `f` for every function call, but continual type checking can add a small runtime overhead when running in eager mode. If `f` is specified, this takes priority, and type checking of the inputs is not used. Furthermore, Ivy avoids importing each of the supported native frameworks for actual type checking. Instead, the types of input arguments are converted to strings for specific keywords search. Importantly, this prevents the need to have all supported native frameworks installed locally.

```
x = ivy.clip(tf.constant([-1, 2.]), 0, 1)
x = ivy.clip(torch.tensor([-1, 2.]), 0, 1)
```

Code snippet 11.5: Examples of type checking.

11.3.5 Global framework setting

A framework can also be set globally for all future function calls. When setting and unsetting globally, the `__dict__` attribute of the framework-specific namespace,

such as `ivy.tensorflow`, is iterated, and each key and value pair is used to directly replace the same key and value pair of the framework-agnostic Ivy namespace `ivy`. This entirely bypasses the framework-agnostic functions, and instead binds all methods of the framework-specific namespace directly to the framework-agnostic namespace. Methods such as `lstm_update` which exist only in framework-agnostic form remain unchanged. Before overwriting, the original framework-agnostic `__dict__` is saved internally, ready for whenever the specific framework is unset again. As long as a framework is set globally, the local framework specification mechanisms are unavailable. This is because the framework-specific functions do not accept an `f` argument or call `get_framework(*args, f=f)`, as the framework agnostic functions do.

Set and unset One option for global framework setting is to use the method `ivy.set_framework()`. Again, the framework can be specified either as a string or as the backend namespace. The backend can then be unset using `ivy.unset_framework()`.

```
ivy.set_framework('tensorflow')
x = ivy.array([-1, 2.])
x = ivy.clip(x, 0, 1)
ivy.unset_framework()
```

Code snippet 11.6: Example of global framework setting.

With statement The framework can also be set globally for a block of code by using the `with` command. In practice, a stack of frameworks is used for global setting and unsetting, allowing multiple blocks to be used in a nested way.

```
with ivy.tensorflow.use:
  x = ivy.array([-1, 2.])
  x = ivy.clip(x, 0.2, 0.8)
```

Code snippet 11.7: Example of global framework block setting.

11.3.6 Ivy Mechanics

Together with Ivy, we provide four libraries of differentiable and framework-agnostic functions. Ivy Mechanics is the lowest level one, as it is used as the building block for Ivy Vision and Ivy Robotics. It mostly focuses on kinematics and transformations of pose representation. For example, in the case of rotations, we provide conversion functions to and from all Euler conventions, quaternions, axis-angle, rotation vectors and rotation matrices.

For an applied example, we consider the case of capturing an omnidirectional

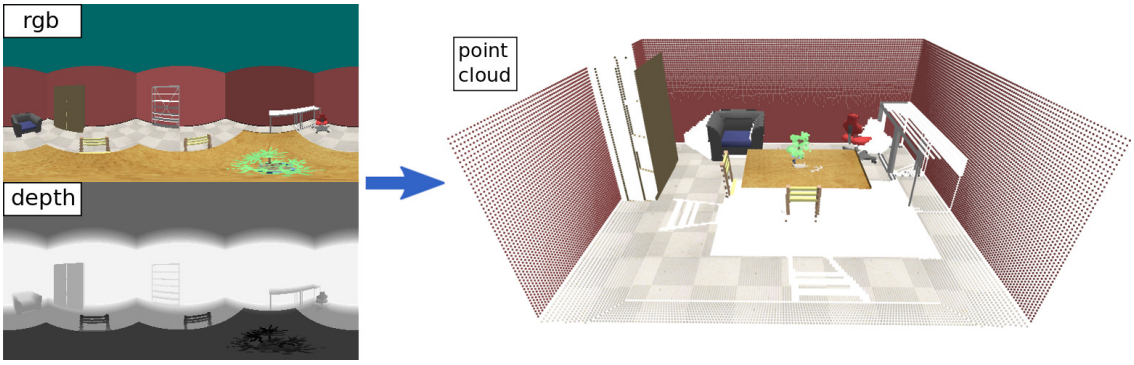


Figure 11.3: Example of image to point cloud transformation. Left: Omnidirectional RGB and depth images, with polar co-ordinate representation. Right: Point cloud following conversion to Cartesian space.

RGBD image, and then using this data to construct a point cloud via the function `polar_coords_to_cartesian_coords(polar_coords)`. Unlike projective images which represent projections onto a flat plane, omnidirectional images represent projections onto a sphere. The pixel indices then correspond to angles in a spherical polar co-ordinate frame. See Figure 11.3 for an example `xyz` point cloud, generated from an omnidirectional image, with the corresponding code sample given below.

```
import ivy_mechanics

pix_per_deg = 2
rgb = sim.omcam.cap()
depth = sim.omdcam.cap()
om_pix = get_pix_coords()
plr_degs = om_pix / pix_per_deg
plr_rads = plr_degs * math.pi / 180
plr = ivy.concatenate([plr_rads, depth], -1)
xyz = ivy_mechanics.polar_coords_to_cartesian_coords(plr)
show_point_cloud(xyz, rgb)
```

Code snippet 11.8: Example of image to point cloud transformation.

11.3.7 Ivy Vision

The Ivy Vision library contains functions that are predominantly targeted at 3D vision, with support for general projections, forward and inverse warping, conversions between depth, flow and pose, voxel grid sampling, and point cloud rendering.

As an example, we use `coords_to_bounding_voxel_grid(coords, voxel_shape_spec)` to render a 100^3 voxel grid, using the point cloud data from Figure 11.3. Figure 11.4 shows the result and the corresponding code is provided below. The voxelization inference time is roughly 15 ms for each backend framework with GPU acceleration.

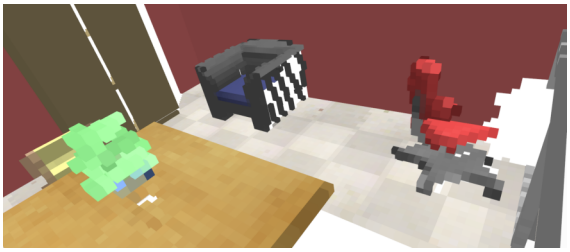


Figure 11.4: Example of scene rendering using the Ivy voxelization function.

```
import ivy_vision

xyz = [depth_to_xyz(dcam.cap()) for dcam in sim.dcams]
xyz = ivy.concatenate(xyz, 1)
voxels = ivy_vision.coords_to_bounding_voxel_grid(xyz, [100] * 3)
render_voxels(voxels)
```

Code snippet 11.9: Example of voxelization inference.

11.3.8 Ivy Robot

The Robot library focuses on robot manipulators, with functions for forward kinematics and spline path parameterization. The library slightly breaks the fully functional Ivy convention, with the use of lightweight manipulator classes. We find this design particularly useful, given the inherent stateful nature of a robot manipulator. For example, intrinsic parameters such as the Denavit–Hartenberg (DH) parameters can be defined once in the constructor, rather than being passed to every function call. We also provide a spline interpolation function, which enables spline sampling when provided with the anchor points.

As an example, we combine this spline function with the manipulator class in order to implement a simplified variant of CHOMP Ratliff et al. (2009). We consider the robot path alone, without accounting for velocity or dynamics, as a 6-DOF spline in joint space. We constrain the start and end joint states, and focus on obstacle avoidant behaviour between these states. We then construct a cost function with two terms. The first one is a collision term which sums the signed-distance-function (SDF) of each sample point with respect to the objects in the scene. The second one sums the length traversed by each link sample point during the motion. This cost function is differentiable and gradient descent allows the path taken by the robot to iteratively improve. An example robot trajectory following convergence is shown in Figure 11.5 together with its corresponding code example below.

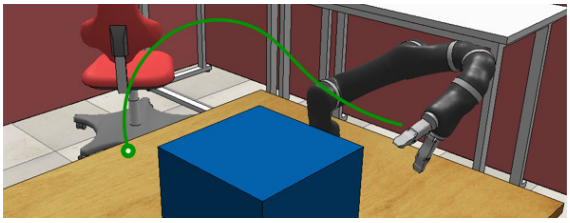


Figure 11.5: Example of robot end-effector motion after convergence.

```
import tensorflow as tf
import ivy_robot

robot = ivy_robot.Manipulator(*get_dh_params())
scene = get_scene_state()
s_pos = get_spline_positions()
crv_degree = get_curve_degree()

for _ in range(10000):
  with tf.GradientTape() as t:
    ancrs = anchor_net()
    j_confs = ivy_robot.interpolate_spline_points(ancrs, s_pos, crv_degree)
    points = robot.sample_links(j_confs, 6)
    sdfs = sdf(points, scene)
    lens = compute_lengths(points)
    cost = sdfs + lens
  vars = anchor_net.trainable_variables
  grads = t.gradient(cost, vars)
  optimizer.apply_gradients(zip(grads, vars))

show_path(points)
```

Code snippet 11.10: Example of robot trajectory optimization.

11.3.9 Ivy Gym

Model-based reinforcement learning relies on an environment model, including forward dynamics and reward function, to optimise a sequence of actions or a policy. When such a model is differentiable, optimisation can use the gradient of the sum of rewards with respect to the sequence of actions or policy parameters, backpropagating through an entire episode. While the model can be learned with a parametric function, it is sometimes desirable to directly use a differentiable environment and focus on other challenges of model-based RL.

The `ivy_gym` library provides a set of such environments, most of which are based on classic control tasks adapted from OpenAI Gym Brockman et al. (2016). Some of those environments can be seen in Figure 11.6.

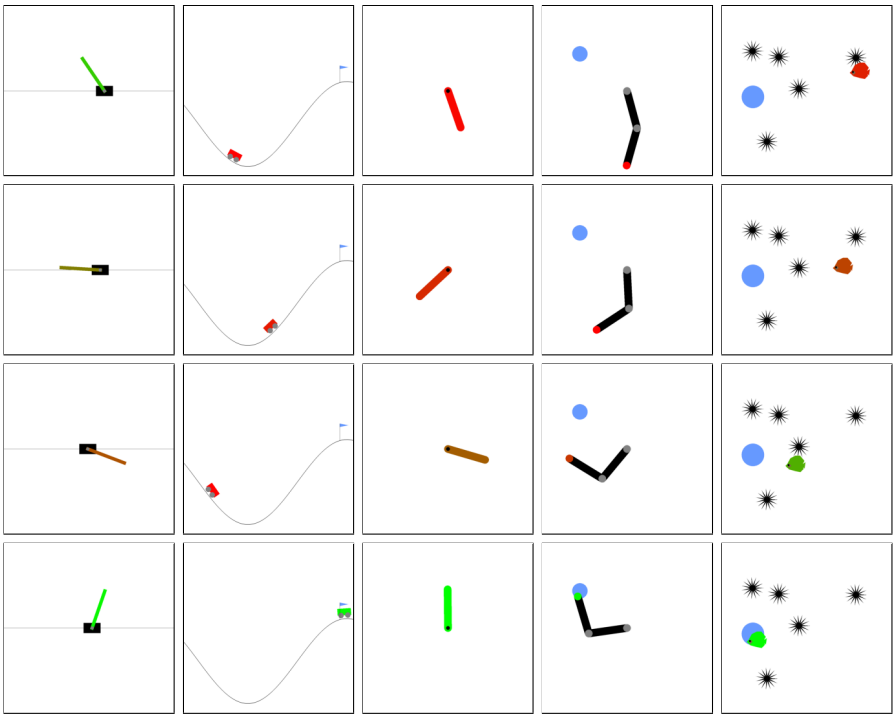


Figure 11.6: Differentiable environments. From left to right: CartPole, MountainCar, Pendulum, Reacher and Swimmer. From top to bottom: different timesteps of optimized episodes.

Dynamics Most classic control environments use clipping and various `if` conditions. For example, the original MountainCar task from Sutton & Barto (2018) uses velocity and position clipping and episode termination around the goal position. On the contrary, the proposed environments have smooth continuous and differentiable dynamics.

```
# MountainCar dynamics
g_acc = g * ivy.cos(3 * x)
x_acc = action * torque_scale - g_acc
x_vel += dt * x_acc
x += dt * x_vel
```

Code snippet 11.11: Dynamics in MountainCar.

Rewards The rewards are also dense, because they are continuous and differentiable everywhere. The values are in the interval $[-1, 1]$ to ensure uniformity between environments, and are represented by an element's color going from red to green when rendering. The original MountainCar task uses a reward of -1 for every time step to motivate episodes to terminate as quickly as possible while our implementation simply rewards the proximity to the goal.

```
# MountainCar reward
dist = (x - goal_x) ** 2
rew = ivy.exp(-5 * dist)
```

Code snippet 11.12: Reward function in MountainCar.

Optimization Since the environments are fully differentiable, a policy or sequence of actions can be used to unroll episodes, and the gradient of the score can be backpropagated in time through the episode to improve actions or policies.

```
ob = env.reset()
score = 0.

with tf.GradientTape() as tape:
  for _ in range(steps):
    ac = policy(ob[None])[0]
    ob, rew, _, _ = env.step(ac)
    score += rew
  loss = -score

grads = tape.gradient(loss, policy.trainable_variables)
optimizer.apply_gradients(zip(grads, policy.trainable_variables))
```

Code snippet 11.13: Example of policy improvement via gradient descent.

11.4 Experiments

11.4.1 Ivy core runtime analysis

In order to assess the overhead introduced by the Ivy abstraction, we perform a runtime analysis for each core function using all possible backend frameworks, and assess how much inference time is consumed by the Ivy abstraction in both eager mode and compiled mode. Ivy code can be compiled using `ivy.compile_fn()`, which wraps the compilation tools from the native frameworks. Our analysis only considers 53 of the 101 core functions implemented at the time of writing, as the remaining 48 Ivy functions incur no overhead for any of the backend frameworks. Each function is called on a CPU with input arrays of size 10000 where possible.

To perform this analysis, we separate the lines of code for each Ivy function into 3 code groups: "backend", "Ivy compilable", and "Ivy eager". "Backend" refers to the native backend operations being wrapped by Ivy. These operations form part of the compilable computation graph. "Ivy compilable" refers to overhead operations added by Ivy, which also form part of the compilable computation graph. For example reshape and transpose operations are sometimes required to unify input-output

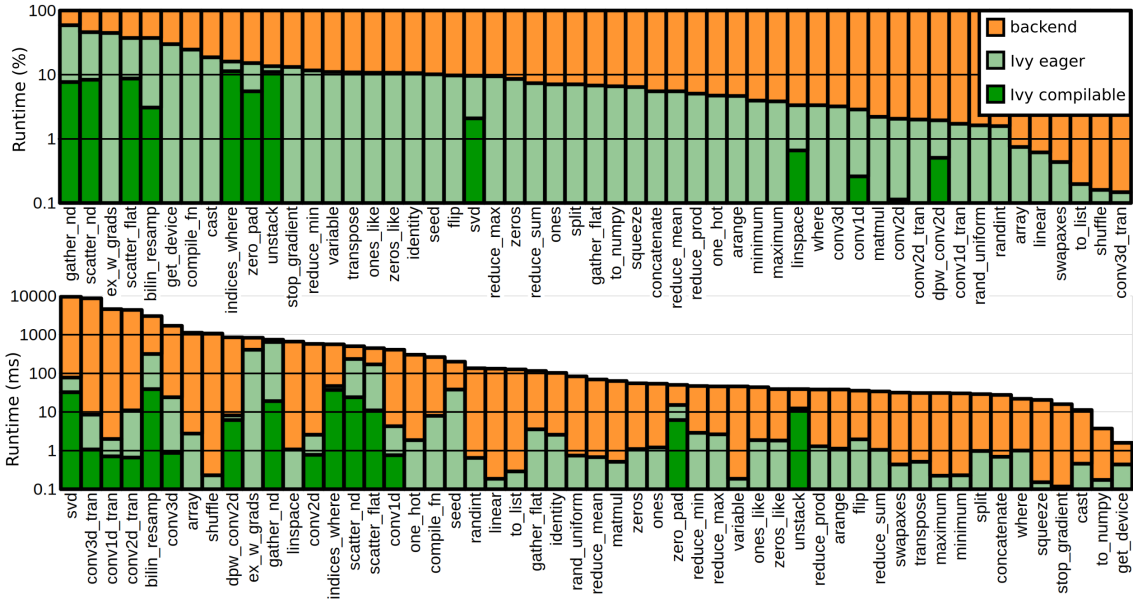


Figure 11.7: Runtimes for each Ivy core method which exhibits some Ivy overhead. The bars are cumulative, with the colors representing the proportion of the runtime consumed by each of the 3 code groups. Note the log scale in both plots.

tensor shapes between frameworks. Finally, "Ivy eager" refers to overhead code which is only executed when running in eager execution mode. If compiled, this code is not run as part of the graph. Examples include inferring the shapes of input tensors via the `shape` attribute, inferring data-types from string input, and constructing new shapes or transpose indices as lists, for defining tensor operations which themselves form part of the compilable computation graph.

A function which consists of `backend` and `Ivy compilable` code is presented below. The transpose operation is necessary to return the output in the expected format.

```
def svd(x, batch_shape=None):
  u, d, v = torch.svd(x)            # backend
  vt = torch.transpose(v, -2, -1)  # Ivy compilable
  return u, d, vt
```

Code snippet 11.14: Example of backend and Ivy compilable code.

A function which consists of `backend` and `Ivy eager` code is presented below. The dictionary lookup is not compiled into the computation graph.

```
def cast(x, dtype_str):
  dtype_val = torch.__dict__[dtype_str]  # Ivy eager
  return x.type(dtype_val)               # backend
```

Code snippet 11.15: Example of backend and Ivy eager code.

We time all Ivy functions in eager mode using the `time.perf_counter()` method on lines of code falling into each of the three code groups. The runtime analysis for each core function averaged across the backend frameworks is presented in Figure 11.7. Without the use of logarithmic scaling, the backend sections would completely dominate the execution times. The functions shown are those that are not direct costless bindings, and include extensions that are not present in some of the supported frameworks. This is for example the case of the gather and scatter functions. In addition, most of the "Ivy compilable" operations produce very little computation time once compiled, and the "Ivy eager" operations disappear completely. Overall, the use of Ivy has a negligible computational cost while providing a very significant gain in terms of portability.

11.5 Discussion

In this chapter, we introduced Ivy, a deep learning library that supports TensorFlow, PyTorch, MXNet, JAX, and NumPy, offering the ability to create framework-agnostic deep learning code. In addition to Ivy, we introduced four libraries for mechanics, 3D vision, robotics, and differentiable environments. We invite developers to join the Ivy community by writing their own functions, layers, and libraries in Ivy, maximizing their direct audience, and helping to accelerate deep learning research by creating sustainable inter-framework code bases.

As for the future vision for Ivy, we will continue to extend the proposed libraries and add new ones for additional research areas. We will also continue to develop Ivy, to remain compatible with the latest developments in deep learning frameworks. We will strive to support the community of open deep learning research through Ivy, and continue to encourage collaboration and contributions from the community.

Chapter 12

Building useful deep reinforcement learning libraries

This chapter is based on the paper "Tonic: A deep reinforcement learning library for fast prototyping and benchmarking" (Pardo, 2020) presented at the NeurIPS 2020 deep RL workshop.

The source code is available at: https://github.com/fabiopardo/tonic.
The benchmark data and trained policies are available at:
https://github.com/fabiopardo/tonic_data.

The main research questions are:

- Can we design a deep RL library to simplify prototyping and benchmarking?
- How to make distributed training compatible with partial-episode bootstrapping?

12.1 Introduction

Deep reinforcement learning research has grown in popularity and been at the heart of many of the recent milestones in artificial intelligence. Interestingly, most of these are based on a number of simple fundamental research ideas that were originally developed independently, such as Q-learning (Watkins & Dayan, 1992), policy gradient (Sutton et al., 2000), and Monte Carlo tree search (Coulom, 2006; Kocsis & Szepesvári, 2006). A large number of research papers introduce novel general purpose ideas for deep reinforcement learning, incorporate them into some existing agents, and evaluate them on a known set of simulated environments against known baselines. Although writing code from scratch has formative qualities, it is generally desirable to use a simple and flexible codebase. While a large number of deep reinforcement learning libraries have been developed over the years, we saw a need for a simple yet modular library designed to quickly implement and evaluate new ideas in an interpretable and fair manner, especially in continuous control.

In this chapter, we introduce Tonic, a library for deep reinforcement learning research, written in Python and supporting both TensorFlow 2 (Abadi et al., 2016) and PyTorch (Paszke et al., 2019). Tonic includes modules such as deep learning models, replay buffers or exploration strategies. Those modules are written to be easily configured and plugged into compatible agents. Furthermore, Tonic implements a number of popular continuous control baseline agents described in Subsection 2.3.1: A2C (Mnih et al., 2016), TRPO (Schulman et al., 2015a), PPO (Schulman et al., 2017), MPO (Abdolmaleki et al., 2018), DDPG (Lillicrap et al., 2015), D4PG (Barth-Maron et al., 2018), TD3 (Fujimoto et al., 2018), and SAC (Haarnoja et al., 2018b). Those agents are written with minimal abstractions to simplify readability and modification, emphasizing core ideas while moving other details into modules and sharing general improvements such as time-awareness and partial-episode bootstrapping, described in Chapter 10, and observation normalization. Tonic also includes three essential scripts to 1) train and test agents in a controlled way 2) plot results against baselines, and 3) play with trained policies. Finally, Tonic includes a large-scale benchmark with training logs and model weights of the baseline agents for 10 seeds on 70 popular environments from OpenAI Gym (Brockman et al., 2016), dm_control (Tassa et al., 2020), and PyBullet (Coumans & Bai, 2016), representing a large and diverse set of domains based on Box2D (Catto, 2011), MuJoCo (Todorov et al., 2012), and Bullet (Coumans, 2010) physics engines.

12.2 Related work

A large number of deep reinforcement learning libraries exist with diverse goals, including large-scale distributed training such as RLlib (Liang et al., 2018), simple and pedagogical code such as Spinning Up (Achiam, 2018), fundamental research in pixel-based domains such as Dopamine (Castro et al., 2018) or based on specific deep learning frameworks including Keras such as Keras-RL (Plappert, 2016), TensorFlow such as OpenAI Baselines (Dhariwal et al., 2017) and TensorFlow Agents (Guadarrama et al., 2018), or PyTorch such as rlpyt (Stooke & Abbeel, 2019) and MushroomRL (D'Eramo et al., 2020). While much effort has been made to build those libraries, we found that there was a need for a simple yet modular codebase designed to quickly try fundamental research ideas and evaluate them in a controlled and fair way, in particular in continuous control domains.

Table 12.1 lists a number of differences between Tonic and other popular existing RL libraries. Repositories that have not received any major update during the past year are marked as not active. With message passing interface (MPI) each worker has its own environment and a copy of the networks, computes gradients based

on its own experience and a synchronous averaging of the gradients is required for each update. This approach does not easily scale to a large number of workers. Asynchronous distributed training does not guarantee reproducible experiments and provides significantly different results on machines with different compute power. Tonic is the only library properly handling timeout terminations and providing three essential scripts to train, plot and play easily.

Library	Active	Frameworks	Modules	Trainer	Distributed	Bench
Tonic (ours)	√	TF 2 and PT	√	√	sync with PEB	large
RLlib	√	TF 1 and PT		√	sync and async	small
Baselines		TF 1			some sync (MPI)	
Stable Baselines		TF 1			some sync (MPI)	
Stable Baselines 3	√	PT				
Spinning Up		TF 1 and PT			some sync (MPI)	small
Acme	√	TF 2 and JAX	√	√	async	
RLgraph		TF 1 and PT	√	√	sync and async	
Coach		TF 1	√	√	sync and async	small

Table 12.1: Comparison between Tonic and other popular existing RL libraries.

12.3 Methods

12.3.1 Library of modules

Many libraries put all the parameters of an experiment at the same level of hierarchy, calling a single agent training function with an environment name and all the relevant parameters. This prevents modularity and readability. On the contrary, Tonic attempts to move the configurable parts into modules as much as possible. This has a number of advantages: 1) it clarifies the parameter targets, avoiding confusing long lists of arguments, 2) different compatible modules with their own specific parameters can be used, and 3) new capabilities can be added to agents, without modifying them.

In Tonic, the configuration of modules takes place in two stages. First, when an experiment is described, agent modules are configured with general-purpose parameters such as hidden layer sizes, exploration noise scale and trace decay used in λ-returns. Then, when the environment is selected, some specific values such as the observation and action space sizes are known, and the modules are finally initialized.

A minimal usage example of Tonic is given below. It shows how an agent, environment, logger, and trainer can be used to create a complete experiment. An illustration of the hierarchy of parameterized modules corresponding to this code snippet is shown in Figure 12.1.

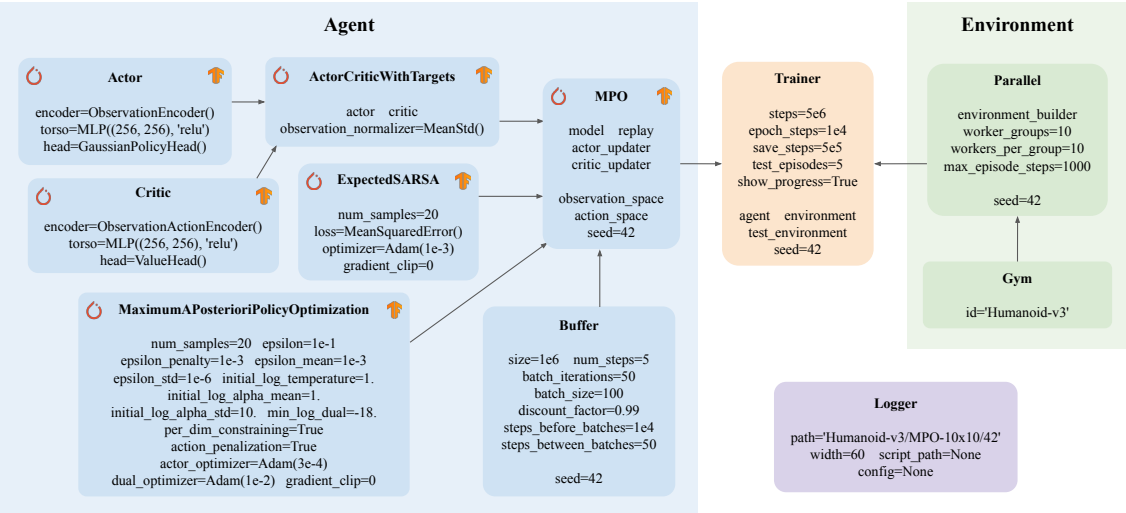


Figure 12.1: A hierarchy of configured modules are used to specify experiments in Tonic. Modules written for both TensorFlow 2 and PyTorch are shown with their respective logos.

```
import tonic.tensorflow

agent = tonic.tensorflow.agents.MPO()

env_fn = lambda: tonic.environments.Gym('Humanoid-v3')
env = tonic.environments.distribute(env_fn, parallel=10, sequential=10)
test_env = tonic.environments.distribute(env_fn)

tonic.logger.initialize('Humanoid-v3/MPO-10x10/42')

trainer = tonic.Trainer(steps=int(1e7), epoch_steps=int(2e4), test_episodes=5)
trainer.initialize(agent, env, test_env, seed=42)
trainer.run()
```

Code snippet 12.1: Usage example of Tonic.

Models For TensorFlow 2 and PyTorch models, smaller modules are assembled. For example, an actor-critic accepts an actor and a critic network. Actors and critics are built with an encoder, a torso and a head module. An encoder processes inputs, for example concatenating observations and actions for an action-dependent critic or normalizing observations using statistics from perceived observations so far. A torso is typically a multilayer perceptron (MLP) or a recurrent network. A head produces the outputs, such as values or distributions.

Replays Different kinds of replays can be used for different types of agents. For example, a traditional `Buffer` can be used to randomly sample past transitions for off-policy training and a `Segment` can be used to store contiguous transitions for an

on-policy agent. Since those replays are configurable modules, they hold parameters like the discount factor or trace decay, and are in charge of producing training batches.

Explorations Different exploration strategies can be used with deterministic actors. Tonic currently includes `Uniform` and `Normal` modules for temporally-uncorrelated action noise and `OrnsteinUhlenbeck` for temporally-correlated action noise.

Updaters Different agents have different ways to update the parameters of their models. An updater typically takes batches of values in input, generates a loss, computes the gradient of this loss with respect to some model parameters and updates those parameters. Some updaters even create new sets of parameters used during optimization such as the dual variables in `MaximumAPosterioriPolicyOptimization`.

Logger The different values generated by the interaction of the agent with the environments and the values generated by the updaters are written in a `csv` file by the logger after each training epoch. When writting those values, outputs are also printed on the terminal in a readable table while a progress bar indicates the remaining time for the current epoch and overall training. The path to the folder containing the logs is usually of the form `'environment/agent/seed/'`. When starting an experiment, the logger can also save the launch script and arguments for future reference and reloading.

12.3.2 Trainer

The `tonic.Trainer` module is in charge of the training loop in Tonic. It takes care of the communication between the agent and the environment, testing the agent on the test environment, logging data via the logger and periodically saving the model parameters in checkpoints for future reload.

Distributed training has been shown to greatly accelerate the training of RL agents with respect to wall clock time (Mnih et al., 2016; Espeholt et al., 2018). Instead of interacting with a single environment at a time, the agent interacts with a set of differently seeded copies of the environment to diversify experience and increase throughput. For simplicity and to ensure reproducibility, Tonic uses a synchronous training loop illustrated in Figure 12.2. At training step t, the trainer first sends a tensor $\boldsymbol{O}_t$ of observations to the agent via the `agent.step` function which returns a tensor $\boldsymbol{A}_t$ of actions and keeps track of some values such as $\boldsymbol{O}_t$ or the log probabilities of the actions. The actions are transmitted to the environment

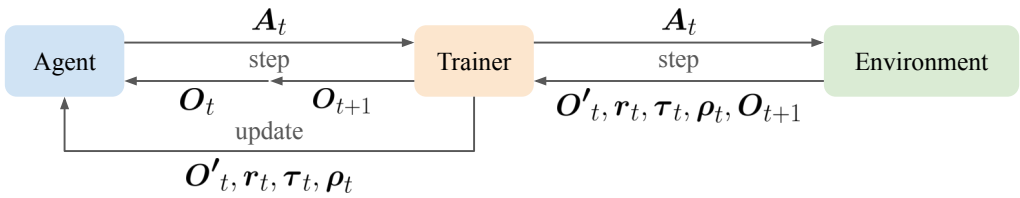


Figure 12.2: Synchronous training loop. At every step, a batch of observations is provided to the agent, which returns a batch of actions in output. Those actions are passed to the environment which returns a batch of transition values used to update the models: next observations, rewards, terminations and resets, plus the batch of next observations used to select the next actions.

module via the `environment.step` function which returns multiple values. First, the ones describing the current transitions caused by the actions $\boldsymbol{A}_t$: the tensor $\boldsymbol{O}'_t$ of next observations, the vectors $\boldsymbol{r}_t$ of rewards, $\boldsymbol{\tau}_t$ of terminations and $\boldsymbol{\rho}_t$ of resets. The terminations indicate true environmental terminations, the ones caused for example by falling on the floor in a locomotion task or reaching a target state. Agents can use those to know when bootstrapping is possible. The resets vector signals the end of episodes, from terminations and timeouts and can be used by agents to know the boundaries of episodes, for example for λ-return calculations. When using non-terminal timeouts, partial-episode bootstrapping, described in Chapter 10, is used to bootstrap from the values in $\boldsymbol{O}'_t$ and it is therefore important to know that a reset happened without an environmental termination. When an environment resets, a new observation is generated and has to be used to select the next action, therefore, the environment also returns a tensor $\boldsymbol{O}_{t+1}$ of observations to use next. For a sub-environment i, $\boldsymbol{o}'^i_t = \boldsymbol{o}^i_{t+1}$ if $\rho^i_t =$ `False`. Finally, the transition values are given to the actor via the `actor.update` function which takes care of registering the transitions in a replay and performing updates, while the new observations $\boldsymbol{O}_{t+1}$ are used to generate the new actions $\boldsymbol{A}_{t+1}$ at the next step.

12.3.3 Agents

Tonic offers a substantial number of continuous control agents. Firstly, a set of random, non-parametric agents is provided. These are mainly for debugging purposes and to serve as primitive baselines. They are the `NormalRandom`, `UniformRandom`, and `OrnsteinUhlenbeck` agents. In addition, for deep reinforcement learning experiments, a number of popular agents, already described in Subsection 2.3.1, are proposed. Some are simple and foundational, such as Advantage Actor-Critic (A2C) (Sutton et al., 2000; Schulman et al., 2015b; Mnih et al., 2016) and Deep Deterministic Policy Gradient (DDPG) (Silver et al., 2014; Lillicrap et al., 2015). Other agents are built on top of these with some more sophisticated components, such as Trust Region Policy

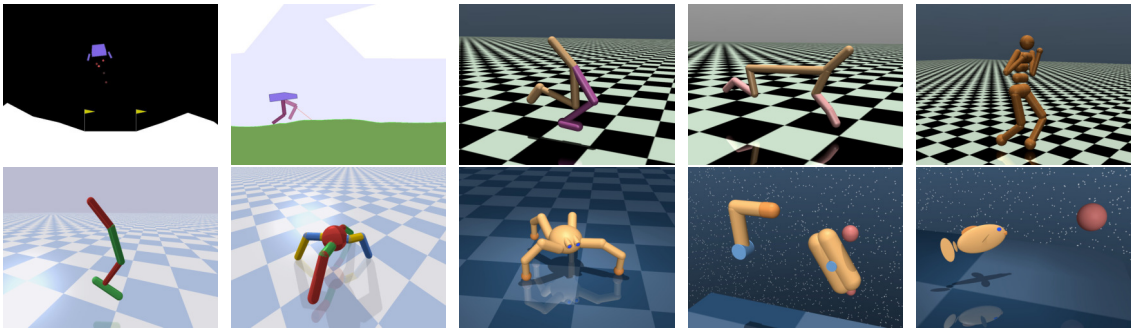


Figure 12.3: Example of environments wrapped by Tonic. From left to right, top to bottom: LunarLanderContinuous-v2, BipedalWalker-v3, Walker2d-v3, HalfCheetah-v3, Humanoid-v3, HopperBulletEnv-v0, AntBulletEnv-v0, quadruped-walk, finger-turn_hard, swimmer-swimmer15.

Optimization (TRPO) (Schulman et al., 2015a) and Proximal Policy Optimization (PPO) (Schulman et al., 2017). Finally, even more complex agents, considered as state of the art are proposed. These include Twin Delayed Deep Deterministic Policy Gradient (TD3) (Fujimoto et al., 2018), Soft Actor-Critic (SAC) (Haarnoja et al., 2018b), Distributed Distributional Deep Deterministic Policy Gradient (D4PG) (Barth-Maron et al., 2018), and Maximum a Posteriori Policy Optimisation (MPO) (Abdolmaleki et al., 2018).

12.3.4 Environments

Compatible suites Tonic includes builders for continuous-control environments from OpenAI Gym (Brockman et al., 2016), dm_control (Tassa et al., 2020), and PyBullet (Coumans & Bai, 2016), representing a large and diverse set of domains based on the Box2D (Catto, 2011), MuJoCo (Todorov et al., 2012), and Bullet (Coumans, 2010) physics engines. A number of compatible environments are pictured in Figure 12.3.

Time limits All of these environments are handled in a unified way that enables the synchronous interaction described in Subsection 12.3.2. Partial-episode bootstrapping, described in Chapter 10, is performed by default for all agents. The `TimeLimit` wrapper is removed from the Gym and PyBullet environments, while in the case of Control Suite environments, task terminations are detected from `task.get_termination(physics)`. Moreover, when `terminal_timeouts` is set to `True`, time-awareness, also described in Chapter 10, can be performed by setting `time_feature` to `True` which enables the use of a `tonic.environments.TimeFeature` wrapper.

Action scaling Agents are all expected to act in a $[-1, 1]^d$ action space where d is the number of action dimensions. This facilitates action noise scaling and learning for agents relying on deterministic policies. To account for the original action spaces of the environments, a `tonic.environments.ActionRescaler` wrapper is used.

Distributed training Finally, for distributed training, the set of environment copies is maintained in parallel groups of sequential workers. Each parallel group is allocated to a different process and communication is done via pipes. Since this communication method adds some time overhead, using multiple sequential environments in each group can increase throughput.

12.3.5 Scripts

The modules and agents described above can easily be used in a standalone experiment Python script or integrated in another codebase. However, for convenience, Tonic includes three essential scripts to take care of the most important things: training, plotting, and playing.

tonic.train The training script is used to launch experiments. Since any agent could be configured with any compatible modules and launched on any configured environment, a simple list of parsed parameters would not give enough flexibility. Therefore, Tonic uses the interpreted nature of the Python language to directly evaluate Python snippets of code passed as strings, describing the agent, the environment and the trainer configurations. The script saves the experiment configuration and automatically initializes the logger to use a path of the form `'environment/agent/seed/'` which will be recognized by the two other scripts.

```
python3 -m tonic.train \
--header "import tonic.torch" \
--agent "tonic.torch.agents.PPO()" \
--environment "tonic.environments.Gym('Ant-v3')" \
--seed 0
```

Code snippet 12.2: Training example.

tonic.plot The plotting script is used to load and display results from multiple experiments together. It expects a list of `csv` or `pkl` files to load data from. Regular expressions like `BipedalWalker-v3/PPO-X/0`, `BipedalWalker-v3/{PPO*,DDPG*}` or `*Bullet*` can be used to point to different sets of logs. Multiple sub-figures are generated, one per environment, aggregating results of agents across runs. The script can be configured in many ways. For example, the figure can be saved in different file

formats such as PDF and PNG. A non-GUI backend such as agg can be used. If the seconds argument is used, plotting is performed regularly in real time, which is very useful for evaluating the performance of an agent during training. The baselines argument can be used to load the included logs from the benchmark, saved in the /data/logs folder at the root of Tonic. For example, --baselines all uses all agents while --baselines A2C PPO TRPO will use logs from A2C, PPO and TRPO. Different parameters allow the user to customize the x and y axes, change the smoothing window size, specify the type of interval shown, display individual runs, select the minimum and maximum values of the x axis, and do many other things. Finally, the legend is shown at the bottom of the figure, regrouping all agents across environments with a mechanism to automatically detect the ideal number of legend columns to use.

```
python3 -m tonic.plot --path Ant-v3 --baselines all
```

Code snippet 12.3: Plotting example.

tonic.play The playing script is used to reinstantiate the environment and agent from an experiment folder, reload weights from a checkpoint and render the policy acting in the environment. The path to the experiment must be specified and a particular checkpoint can be chosen. While rendering the policy interacting with the environment, the episode lengths, scores, min and max rewards are printed on the terminal. Gym environments are simply rendered while PyBullet and dm_control viewers allow users to add perturbations to the bodies in the simulation.

```
python3 -m tonic.play --path BipedalWalker-v3/PPO/0
```

Code snippet 12.4: Playing example.

Adding new modules, agents and environments When using tonic.train, new components can be added using the header field which is evaluated first. For example, if the components are installed or accessible from the current directory, they can directly be imported. If necessary, the path to the required files can also be added to sys.path before importing them. Finally, for OpenAI Gym, PyBullet and dm_control environments, it is recommended to register new tasks at import time in the packages themselves. For example the __init__.py at the root of a package containing new tasks could use register(id='Cat-v0', entry_point=CatEnv) for Gym and PyBullet, and suite._DOMAINS['cat'] = cat_tasks for Control Suite.

```
python3 -m tonic.train \
--header "import animal_envs, tonic.tensorflow" \
--environment "tonic.environments.Gym('Cat-v0')" \
--agent "tonic.tensorflow.agents.TD3()" \
--seed 0
```

Code snippet 12.5: Training with a custom environment.

12.4 Experiments

12.4.1 Large-scale benchmark

When evaluating novel ideas in the literature, it is sometimes difficult to measure the significance of results as baselines can be poorly tuned or evaluated in unfair conditions. Benchmarks of popular RL agents on popular RL environments (Duan et al., 2016; Huang et al., 2020a) evaluated in identical conditions are essential to provide reliable lower bounds in fundamental research. Tonic contains a large-scale benchmark of the 8 provided deep RL agents on 70 popular continuous-control tasks: 17 from OpenAI Gym (2 classic control, 3 Box2D, 12 MuJoCo), 10 from PyBullet and 43 from the benchmark subset of the DeepMind Control Suite (Tassa et al., 2018) in dm_control. The exact same 10 seeds (0, 1, 2, ..., 9) are used for all agents with default parameters on all environments with single-worker training (not distributed). D4PG was only run on dm_control environments because known reward boundaries are required for distributional value functions. Therefore, the total number of runs contained in the benchmark is $70 \times 10 \times 7 + 43 \times 10 = 5330$. These runs were all generated with `tonic.tensorflow`.

For each agent, 5 test episodes are collected after each training epoch and averaged. The benchmark results can be seen in Figure 12.4. Environments from OpenAI Gym (names starting with an uppercase) and PyBullet (names with "PybulletEnv") are mostly "solved" with the best agents getting scores similar to the best performances reported in the literature. However, many environments from dm_control seem much harder to solve. Most results reported in the literature for those environments actually use distributed training and more training steps. Moreover, better hyper parameters could certainly be found for those agents, and especially better ones for each environment specifically.

12.4.2 Speed comparison between frameworks

A significant difference in training speed was observed between the Tensorflow 2 and PyTorch implementations of the same agents. To measure this difference, an

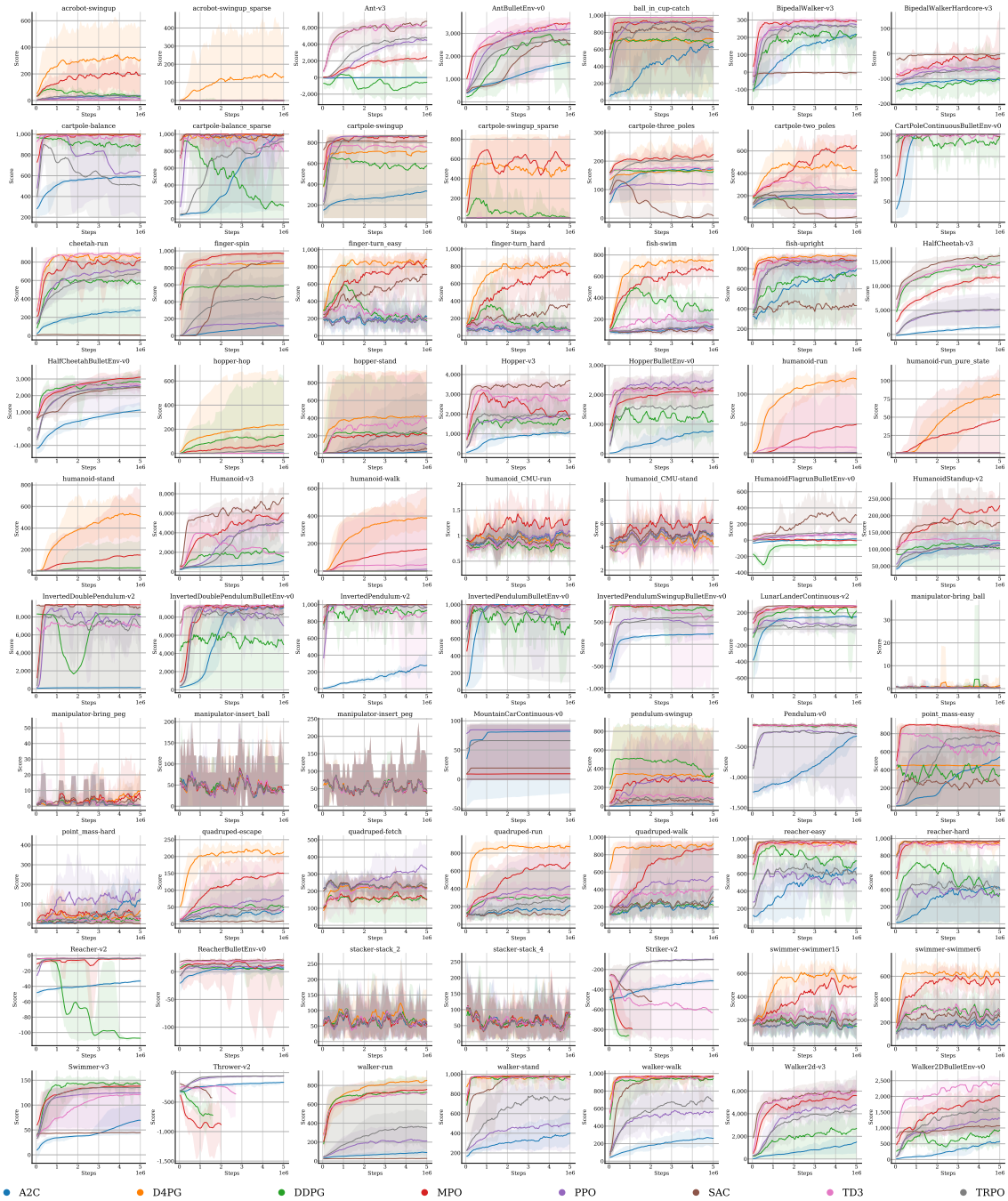


Figure 12.4: Benchmark results on 70 tasks. For each agent, 5 test episodes are collected after each training epoch and averaged. The solid lines represent the average over 10 runs for each agent. The [minimum, maximum] range is shown with transparent areas and a sliding window of size 5 is used for smoothing. A large palette of environments are represented across the supported domains. The best performing agents are mostly TD3, SAC, MPO and D4PG but significant variations exist for each agent.

experiment involving each of the 8 agents on walker-walk was used. Agents were trained for 1 million steps, using the same parameters as for the benchmark. The

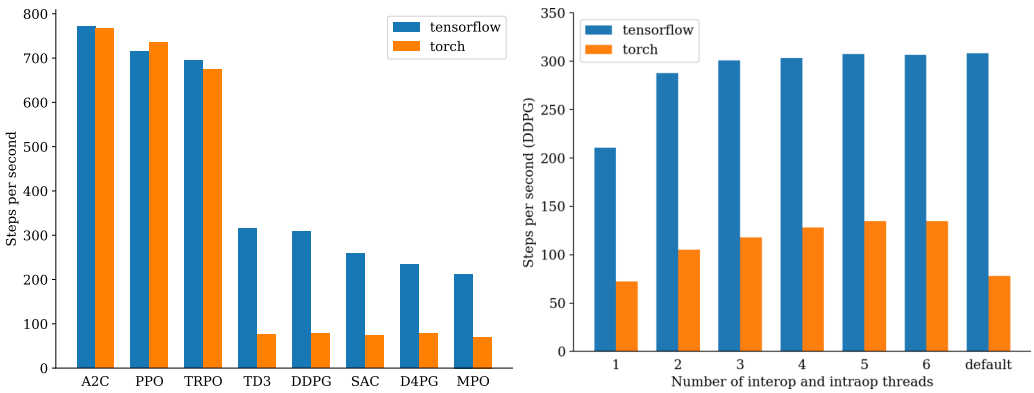


Figure 12.5: Speed comparison between TensorFlow 2 and PyTorch agents on walker-walk. The figure on the left used the default number of threads while the figure on right shows the impact of setting the number of interop and intraop threads for DDPG.

time spent to run the last 250,000 steps is used to measure the indicated average number of steps per second. The agents were processed in turn on the same 6-core processor running at 3.8 GHz without GPU. Figure 12.5 shows the results of this comparison. On off-policy agents, TensorFlow 2 is significantly faster even after tuning the number of threads used. The difference between the two frameworks might be due to a more efficient graph tracing mechanism provided by TensorFlow's `tf.function` decorator.

12.4.3 Comparison with Spinning Up

To prove that the results for A2C, TRPO, PPO, DDPG, TD3, and SAC can be used as valid baselines, another benchmark was generated with TensorFlow 1 implementations of those agents from Spinning Up (Achiam, 2018). The library was slightly modified to use a test environment for each agent, a frequency and number of test episodes and seeds identical to the ones used in the Tonic benchmark, and VPG was renamed A2C. Results on the original 5 environments used in the benchmark of this library can be seen in Figure 12.6.

The results from Spinning Up are compatible with the ones found on the website[1]. The agents from Tonic perform significantly better on four of the five environments. This difference can be explained by some of the improvements in Tonic, such as observation normalization, partial-episode bootstrapping, and action-scaling. Interestingly, Spinning Up partially implements PEB for the off-policy methods only, by ignoring environmental terminations at timeouts, an heuristic that is incorrect when true terminations occur simultaneously.

[1]https://spinningup.openai.com/en/latest/spinningup/bench.html

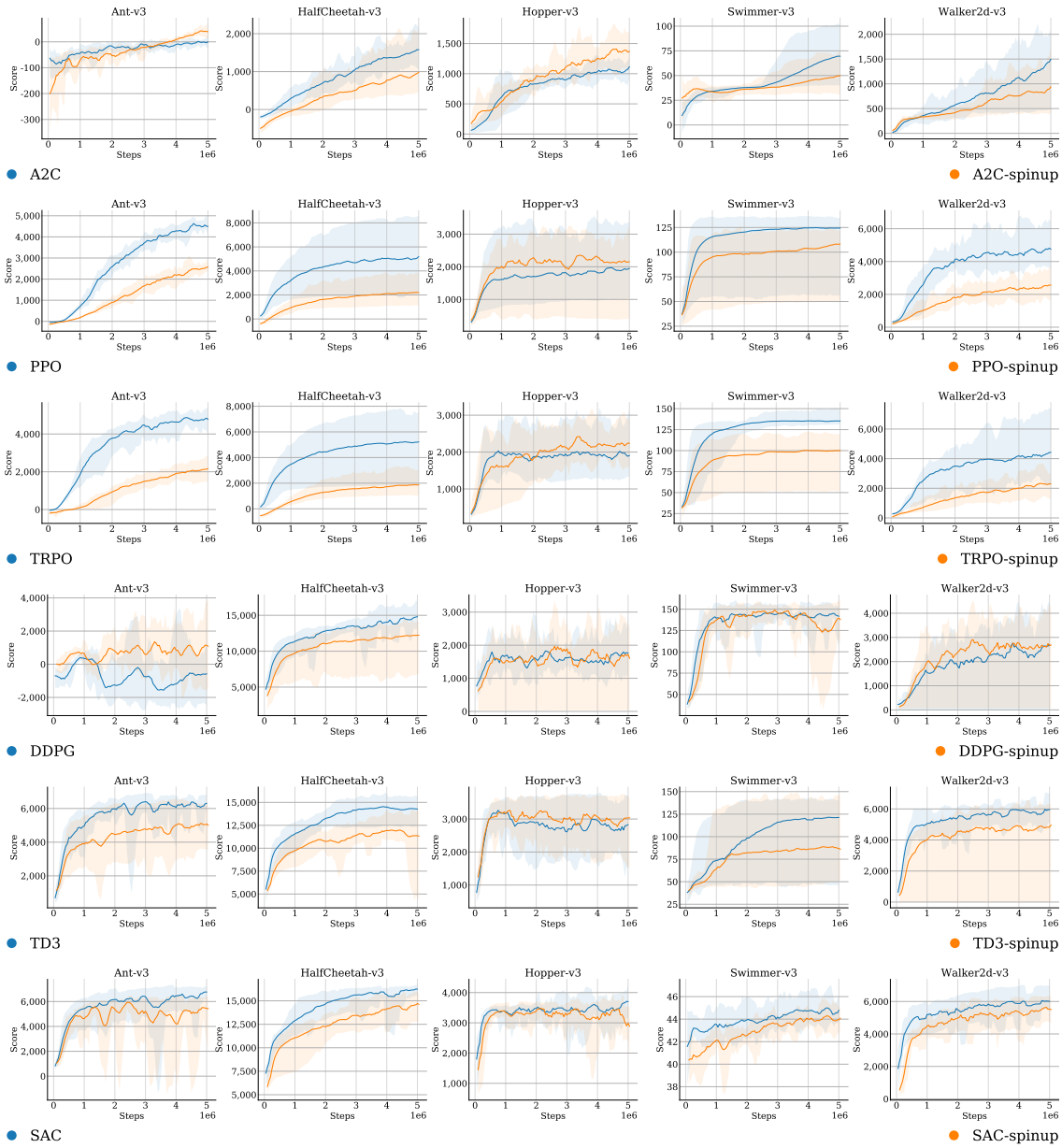


Figure 12.6: Performance comparison with Spinning Up using the same training, evaluation and agent parameters.

12.4.4 Ablations and variants

To measure the effectiveness of partial-episode bootstrapping and observation normalization, PPO was trained with different variants. The results can be seen in Figure 12.7. Observation normalization appears to significantly accelerate learning and PEB (non-terminal timeouts) is best, while time features are needed to better account for terminal timeouts as discussed in Chapter 10.

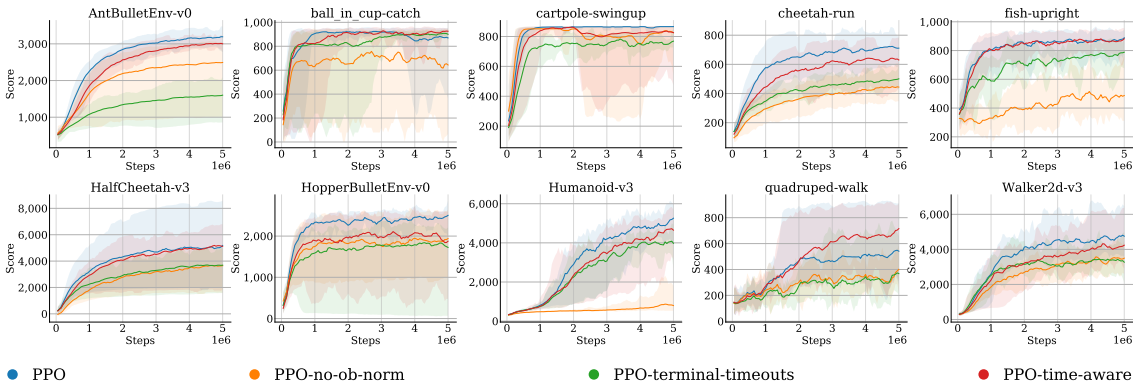


Figure 12.7: Efficiency of observation normalization and non-terminal timeouts. PPO uses the default configuration with observation normalization and non-terminal timeouts. PPO-no-ob-norm is identical but without observation-normalization. PPO-terminal-timeouts is identical to PPO but with environmental terminations at timeouts as is the case originally for the supported environments. PPO-time-aware is identical to PPO-terminal-timeouts but adds the remaining time as an observed feature.

12.4.5 Prototyping and benchmarking a novel agent

To demonstrate how Tonic can accelerate the development and the evaluation of ideas, a new agent called TD4, combining features of TD3 and D4PG is proposed. The code required to implement this agent contains three elements. The first one is a new model based on `ActorTwinCriticWithTargets` with a distributional value head for the twin critic.

```
def default_model():
  return models.ActorTwinCriticWithTargets(
    actor=models.Actor(
      encoder=models.ObservationEncoder(),
      torso=models.MLP((256, 256), 'relu'),
      head=models.DeterministicPolicyHead()),
    critic=models.Critic(
      encoder=models.ObservationActionEncoder(),
      torso=models.MLP((256, 256), 'relu'),
      head=models.DistributionalValueHead(-150., 150., 51)),
    observation_normalizer=normalizers.MeanStd())
```

Code snippet 12.6: TD4 model.

The second element is a new updater similar to `TwinCriticDeterministicQLearning` but adapted to use a pair of critics.

```
class TwinCriticDistributionalDeterministicQLearning:
  def __init__(self, optimizer=None, target_action_noise=None, gradient_clip=0):
    self.optimizer = optimizer or \
      tf.keras.optimizers.Adam(lr=1e-3, epsilon=1e-8)
    self.target_action_noise = target_action_noise or \
      updaters.TargetActionNoise(scale=0.2, clip=0.5)
    self.gradient_clip = gradient_clip

  def initialize(self, model):
    self.model = model
    variables_1 = self.model.critic_1.trainable_variables
    variables_2 = self.model.critic_2.trainable_variables
    self.variables = variables_1 + variables_2

  @tf.function
  def __call__(self, observations, actions, next_observations, rewards, discounts):
    next_actions = self.model.target_actor(next_observations)
    next_actions = self.target_action_noise(next_actions)
    next_value_distributions_1 = self.model.target_critic_1(
      next_observations, next_actions)
    next_value_distributions_2 = self.model.target_critic_2(
      next_observations, next_actions)

    values = next_value_distributions_1.values
    returns = rewards[:, None] + discounts[:, None] * values
    targets_1 = next_value_distributions_1.project(returns)
    targets_2 = next_value_distributions_2.project(returns)
    next_values_1 = next_value_distributions_1.mean()
    next_values_2 = next_value_distributions_2.mean()
    twin_next_values = tf.concat(
      [next_values_1[None], next_values_2[None]], axis=0)
    indices = tf.argmin(twin_next_values, axis=0, output_type=tf.int32)
    twin_targets = tf.concat([targets_1[None], targets_2[None]], axis=0)
    batch_size = tf.shape(observations)[0]
    indices = tf.stack([indices, tf.range(batch_size)], axis=-1)
    targets = tf.gather_nd(twin_targets, indices)

    with tf.GradientTape() as tape:
      value_distributions_1 = self.model.critic_1(observations, actions)
      losses_1 = tf.nn.softmax_cross_entropy_with_logits(
        logits=value_distributions_1.logits, labels=targets)
      value_distributions_2 = self.model.critic_2(observations, actions)
      losses_2 = tf.nn.softmax_cross_entropy_with_logits(
        logits=value_distributions_2.logits, labels=targets)
      loss = tf.reduce_mean(losses_1) + tf.reduce_mean(losses_2)

    gradients = tape.gradient(loss, self.variables)
    if self.gradient_clip > 0:
      gradients = tf.clip_by_global_norm(gradients, self.gradient_clip)[0]
    self.optimizer.apply_gradients(zip(gradients, self.variables))

    return dict(loss=loss)
```

Code snippet 12.7: TD4 critic updater.

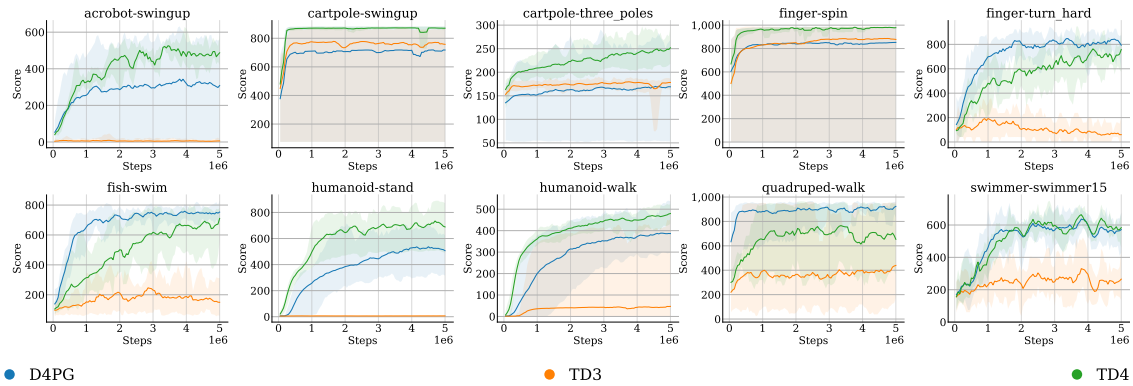


Figure 12.8: Evaluation of the proposed TD4 agent. The performance is significantly better on 6 of the 10 tasks, demonstrating that the agent successfully combines features from D4PG and TD3.

The third element is the agent itself, based on `TD3` and `D4PG` and using the two previous elements.

```
class TD4(agents.TD3):
  def __init__(
    self, model=None, replay=None, exploration=None, actor_updater=None,
    critic_updater=None, delay_steps=2
  ):
    model = model or default_model()
    replay = replay or replays.Buffer(num_steps=5)
    actor_updater = actor_updater or \
      updaters.DistributionalDeterministicPolicyGradient()
    critic_updater = critic_updater or \
      TwinCriticDistributionalDeterministicQLearning()
    super().__init__(
      model=model, replay=replay, exploration=exploration,
      actor_updater=actor_updater, critic_updater=critic_updater,
      delay_steps=delay_steps)
```

Code snippet 12.8: TD4 agent.

The new agent can simply be integrated into Tonic when training, using the `header` field, and its performance can directly be compared to the baseline scores of D4PG and TD3.

Training can be started using:

```
python3 -m tonic.train \
--header "import td4, tonic.tensorflow" \
--environment "tonic.environments.ControlSuite('humanoid-walk')" \
--agent "td4.TD4()" \
--seed 0
```

Code snippet 12.9: Training TD4.

The agent can be evaluated against D4PG and TD3 using:

```
python3 -m tonic.plot --path humanoid-walk --baselines D4PG TD3
```

Code snippet 12.10: Benchmarking TD4.

The results on 10 tasks are shown in Figure 12.8 and demonstrate that TD4 is an excellent agent combining advantages from TD3 and D4PG and was particularly simple to implement and evaluate.

12.5 Discussion

This chapter introduced Tonic, a library designed for fast prototyping and benchmarking of deep reinforcement learning algorithms. The library contains numerous configurable modules and agents. It also provides support for the two most popular deep learning frameworks and the three most popular continuous control environment suites. Its three essential scripts for training and evaluating agents, and its large-scale benchmark can facilitate experimentation to a large extent. This is the last work presented in this thesis, and it encompasses much of the knowledge for designing and training agents, described in this thesis.

Future work will include support for discrete action spaces and pixel-based observations, better handling of dictionary-based observations, benchmark results with improved hyperparameters, new modules, and agents. In particular, some of the new agents will rely on discretization of continuous-action spaces as this alternative has proven to be competitive with continuous-control methods as shown in Chapter 4. Hopefully researchers will use Tonic, contribute to it and will find easier to release the source code of their own work.

Chapter 13

Conclusion and future work

In this thesis, I have given a general description of what deep reinforcement learning research is, listing a subset of the myriad of elements that constitute experiments, and have proposed to group them into three parts: agents, environments, and infrastructures. To illustrate them, I have also presented seven research papers, giving a representative sample of the type of challenges, both theoretical and experimental, that can be envisaged in the field. Special care has also been taken to open sourcing code and data, which I hope will benefit a large number of researchers.

Elements of agent design were discussed in Part I, but demonstrated in various works throughout the entire thesis: from neural architectures and update rules in Chapter 4, Chapter 5, and Chapter 7, to hierarchy in Chapter 5 and Chapter 7, through exploration strategies in Chapter 5, and skill discovery in Chapter 7. Most of the time I spent on my research was actually invested in agent design, although the majority of these projects were not included in the thesis. In fact, the topics that have occupied my mind the most have been hierarchy, planning, and skill representations. In particular, I think that planning should be flexible, involve varying degrees of skill composition, take the form of a tree search when necessary, but also allow for more dynamic connections. It should involve analogies to past experiences, imagination, and temporal abstraction. In addition, another topic I would like to continue studying is exploration. I believe that the ideal exploration strategy should be learned and should give credit to exploratory actions that allowed learning to progress over the course of an agent's life.

Elements of environment design were discussed in Part II, and demonstrated in terms of model creation in Chapter 8, action space choices in Chapter 4, Chapter 7 and Chapter 8, observation space choices in Chapter 7, Chapter 8 and Chapter 10, terminations and resets in Chapter 10 and Chapter 12, and more broadly tasks in Chapter 7, Chapter 8 and Chapter 10. During my research, I realized the importance of thinking about environments to use even throughout the process of designing

agents. For example, custom tasks can be useful to study a specific feature of an agent as was done in Chapter 4 when varying the number of degrees of freedom of a robot arm, or in Chapter 10 with toy problems to illustrate our claims. However, a drawback of creating such environments is that they are inherently biased toward confirming the hypotheses we are testing. On the other hand, using only well-studied benchmarks to focus on the effectiveness of proposed methods against known baselines is a less biased process and allows us to better evaluate the effectiveness of the approaches, but in some sense also narrows down the narration and limits our creativity. Ideally, both should be used. I am also particularly excited about recent advances in transfer from simulation to robots. I believe that for much of the training, simulation is well justified as it allows to collect orders of magnitude more samples and simulations are becoming increasingly more realistic. This is not to say that learning should not be pursued on robots at all. What matters is to keep pushing the limits of what is possible. Furthermore, we should not forget that animals do not learn from randomly initiated neural networks, but inherit intrinsic knowledge that is the result of millions of years of natural selection, a form of learning.

Finally, elements of infrastructure design were discussed in Part III. These include the choice of deep learning framework in Chapter 11 and Chapter 12, scheduling of tasks in Chapter 7 and scheduling of resets in Chapter 10 and Chapter 12, distributed training in Chapter 12, and transfer in Chapter 5 and Chapter 7. We also discussed the importance of reproducibility, robustness, fair comparison with properly tuned baselines, and open sourcing code in Chapter 12. During my PhD, I used several deep reinforcement learning libraries, but I was never satisfied with some elements such as ease of prototyping, modularity, visualization of results, time limits management, available agents, and benchmarks. I tried several times to write a library of my own. It took a pandemic to convince me to step back from traditional deep reinforcement learning research and develop a tool that fits my needs and has the potential to help other researchers in the field. I will continue to contribute to its development as long as it remains useful.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A system for large-scale machine learning. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

Abdolmaleki, A., Springenberg, J. T., Tassa, Y., Munos, R., Heess, N., and Riedmiller, M. Maximum a posteriori policy optimisation. *arXiv preprint arXiv:1806.06920*, 2018.

Abdolmaleki, A., Huang, S., Hasenclever, L., Neunert, M., Song, F., Zambelli, M., Martins, M., Heess, N., Hadsell, R., and Riedmiller, M. A distributional view on multi-objective policy optimization. *International Conference on Machine Learning (ICML)*, 2020.

Abdrashitov, R., Bang, S., Levin, D. I., Singh, K., and Jacobson, A. Interactive modelling of volumetric musculoskeletal anatomy. *arXiv preprint arXiv:2106.05161*, 2021.

Achiam, J. Spinning up in deep reinforcement learning, 2018. https://github.com/openai/spinningup.

Achiam, J., Edwards, H., Amodei, D., and Abbeel, P. Variational option discovery algorithms. *arXiv preprint arXiv:1807.10299*, 2018.

Akimov, D. Distributed soft actor-critic with multivariate reward representation and knowledge distillation. *arXiv preprint arXiv:1911.13056*, 2019.

Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Bleecher Snyder, J., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., and Zhang, Y. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*, 2016.

Alexander, R. M., Maloiy, G., Njau, R., and Jayes, A. Mechanics of running of the ostrich (Struthio camelus). *Journal of Zoology*, 1979.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., and Zaremba, W. Hindsight experience replay. *arXiv preprint arXiv:1707.01495*, 2017.

Angles, B., Rebain, D., Macklin, M., Wyvill, B., Barthe, L., Lewis, J., Von Der Pahlen, J., Izadi, S., Valentin, J., Bouaziz, S., and Tagliasacchi, A. VIPER: Volume invariant position-based elastic rods. *Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2019.

Apgar, T., Clary, P., Green, K., Fern, A., and Hurst, J. W. Fast online trajectory optimization for the bipedal robot Cassie. *Robotics: Science and Systems*, 2018.

Bacon, P.-L., Harb, J., and Precup, D. The option-critic architecture. *AAAI Conference on Artificial Intelligence*, 2017.

Badia, A. P., Sprechmann, P., Vitvitskyi, A., Guo, D., Piot, B., Kapturowski, S., Tieleman, O., Arjovsky, M., Pritzel, A., Bolt, A., and Blundell, C. Never give up: Learning directed exploration strategies. *arXiv preprint arXiv:2002.06038*, 2020.

Bakker, B. and Schmidhuber, J. Hierarchical reinforcement learning based on subgoal discovery and subpolicy specialization. *International Conference on Intelligent Autonomous Systems (IAS)*, 2004.

Baranes, A. and Oudeyer, P.-Y. Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems (RAS)*, 2013.

Barreto, A., Borsa, D., Hou, S., Comanici, G., Aygün, E., Hamel, P., Toyama, D., Hunt, J., Mourad, S., Silver, D., and Precup, D. The option keyboard: Combining skills in reinforcement learning. *arXiv preprint arXiv:2106.13105*, 2021.

Barth-Maron, G., Hoffman, M. W., Budden, D., Dabney, W., Horgan, D., Tb, D., Muldal, A., Heess, N., and Lillicrap, T. Distributed distributional deterministic policy gradients. *arXiv preprint arXiv:1804.08617*, 2018.

Barto, A. G. Intrinsic motivation and reinforcement learning. *Intrinsically Motivated Learning in Natural and Artificial Systems*, 2013.

Bellemare, M., Srinivasan, S., Ostrovski, G., Schaul, T., Saxton, D., and Munos, R. Unifying count-based exploration and intrinsic motivation. *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research (JAIR)*, 2013.

Bellemare, M. G., Dabney, W., and Munos, R. A distributional perspective on reinforcement learning. *International Conference on Machine Learning (ICML)*, 2017.

Bergamin, K., Clavet, S., Holden, D., and Forbes, J. R. DReCon: Data-driven responsive control of physics-based characters. *ACM Transactions On Graphics (TOG)*, 2019.

Bertsekas, D. Dynamic programming and optimal control, 2012.

Bertsekas, D. P. and Tsitsiklis, J. N. *Neuro-dynamic programming.* Athena Scientific, 1996.

Blundell, C., Uria, B., Pritzel, A., Li, Y., Ruderman, A., Leibo, J. Z., Rae, J., Wierstra, D., and Hassabis, D. Model-free episodic control. *arXiv preprint arXiv:1606.04460*, 2016.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: Composable transformations of Python+NumPy programs, 2018. https://github.com/google/jax.

Brafman, R. I. and Tennenholtz, M. R-MAX - A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research (JMLR)*, 2002.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

Broughton, M., Verdon, G., McCourt, T., Martinez, A., Yoo, J., Isakov, S., Massey, P., Niu, Y., Halavati, R., Peters, E., Leib, M., Skolik, A., Streif, M., Von Dollen, D., Mcclean, J., Boixo, S., Bacon, D., Ho, A., Neven, H., and Mohseni, M. TensorFlow Quantum: A software framework for quantum machine learning. *arXiv preprint arXiv:2003.02989*, 2020.

Bulat, M., Can, N. K., Arslan, Y. Z., and Herzog, W. Musculoskeletal simulation tools for understanding mechanisms of lower-limb sports injuries. *Current Sports Medicine Reports*, 2019.

Burda, Y., Edwards, H., Pathak, D., Storkey, A., Darrell, T., and Efros, A. A. Large-scale study of curiosity-driven learning. *arXiv preprint arXiv:1808.04355*, 2018a.

Burda, Y., Edwards, H., Storkey, A., and Klimov, O. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018b.

Buss, S. R. Introduction to inverse kinematics with Jacobian transpose, pseudoinverse and damped least squares methods. *IEEE Journal of Robotics and Automation*, 2004.

Böhmer, C., Prevoteau, J., Duriez, O., and Abourachid, A. Gulper, ripper and scrapper: Anatomy of the neck in three species of vultures. *Journal of Anatomy*, 2019.

Cabi, S., Colmenarejo, S. G., Hoffman, M. W., Denil, M., Wang, Z., and Freitas, N. The intentional unintentional agent: Learning to solve many continuous control tasks simultaneously. *Conference on Robot Learning (CoRL)*, 2017.

Cassirer, A., Barth-Maron, G., Brevdo, E., Ramos, S., Boyd, T., Sottiaux, T., and Kroiss, M. Reverb: A framework for experience replay. *arXiv preprint arXiv:2102.04736*, 2021.

Castro, P. S., Moitra, S., Gelada, C., Kumar, S., and Bellemare, M. G. Dopamine: A research framework for deep reinforcement learning. *arXiv preprint arXiv:1812.06110*, 2018.

Catto, E. Box2D: A 2D physics engine for games, 2011. https://github.com/erincatto/box2d.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chentanez, N., Müller, M., Macklin, M., Makoviychuk, V., and Jeschke, S. Physics-based motion capture imitation with deep reinforcement learning. *Conference on Motion, Interaction and Games (MIG)*, 2018.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

Chollet, F. et al. Keras, 2015. https://keras.io.

Colas, C., Sigaud, O., and Oudeyer, P.-Y. GEP-PG: Decoupling exploration and exploitation in deep reinforcement learning algorithms. *International Conference on Machine Learning (ICML)*, 2018.

Coste, C. A., Bergeron, V., Berkelmans, R., Martins, E. F., Fornusek, C., Jetsada, A., Hunt, K. J., Tong, R., Triolo, R., and Wolf, P. Comparison of strategies and performance of functional electrical stimulation cycling in spinal cord injury pilots for competition in the first ever CYBATHLON. *European Journal of Translational Myology*, 2017.

Cotton, S., Olaru, I. M. C., Bellman, M., van der Ven, T., Godowski, J., and Pratt, J. FastRunner: A fast, efficient and robust bipedal robot. concept and planar simulation. *IEEE International Conference on Robotics and Automation*, 2012.

Coulom, R. Efficient selectivity and backup operators in Monte-Carlo tree search. *International Conference on Computers and Games (ICCG)*, 2006.

Coumans, E. Bullet physics engine, 2010. http://pybullet.org.

Coumans, E. and Bai, Y. PyBullet: A Python module for physics simulation for games, robotics and machine learning, 2016.

Cuff, A. R., Daley, M. A., Michel, K. B., Allen, V. R., Lamas, L. P., Adami, C., Monticelli, P., Pelligand, L., and Hutchinson, J. R. Relating neuromuscular control to functional anatomy of limb muscles in extant archosaurs. *Journal of Morphology*, 2019.

Cully, A., Clune, J., Tarapore, D., and Mouret, J.-B. Robots that can adapt like animals. *Nature*, 2015.

Dayan, P. and Hinton, G. Feudal reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 1992.

Delp, S. L., Anderson, F. C., Arnold, A. S., Loan, P., Habib, A., John, C. T., Guendelman, E., and Thelen, D. G. OpenSim: Open-source software to create and analyze dynamic simulations of movement. *IEEE Transactions on Biomedical Engineering*, 2007.

D'Eramo, C., Tateo, D., Bonarini, A., Restelli, M., and Peters, J. MushroomRL: Simplifying reinforcement learning research. *arXiv preprint arXiv:2001.01102*, 2020. https://github.com/mushroomrl/mushroom-rl.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. OpenAI Baselines, 2017.

Dietterich, T. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research (JAIR)*, 2000.

Dillon, J. V., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., and Saurous, R. A. TensorFlow distributions. *arXiv preprint arXiv:1711.10604*, 2017.

Dosovitskiy, A., Fischer, P., Ilg, E., Hausser, P., Hazirbas, C., Golkov, V., Van Der Smagt, P., Cremers, D., and Brox, T. FlowNet: Learning optical flow with convolutional networks. *IEEE International Conference on Computer Vision (ICCV)*, 2015.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. Benchmarking deep reinforcement learning for continuous control. *International Conference on Machine Learning (ICML)*, 2016.

Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research (JMLR)*, 2011.

Dulac-Arnold, G., Evans, R., van Hasselt, H., Sunehag, P., Lillicrap, T., Hunt, J., Mann, T., Weber, T., Degris, T., and Coppin, B. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

Dzemski, G. and Christian, A. Flexibility along the neck of the ostrich (Struthio camelus) and consequences for the reconstruction of dinosaurs with extreme neck length. *Journal of Morphology*, 2007.

Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K. O., and Clune, J. First return, then explore. *Nature*, 2021.

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. *International Conference on Machine Learning (ICML)*, 2018.

Eysenbach, B., Gupta, A., Ibarz, J., and Levine, S. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

Faloutsos, P., Van de Panne, M., and Terzopoulos, D. Composable controllers for physics-based character animation. *Computer Graphics and Interactive Techniques (SIGGRAPH)*, 2001.

Fey, M. and Lenssen, J. E. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428*, 2019.

Florensa, C., Duan, Y., and Abbeel, P. Stochastic neural networks for hierarchical reinforcement learning. *arXiv preprint arXiv:1704.03012*, 2017a.

Florensa, C., Held, D., Wulfmeier, M., Zhang, M., and Abbeel, P. Reverse curriculum generation for reinforcement learning. *Conference on Robot Learning (CoRL)*, 2017b.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., and Legg, S. Noisy networks for exploration. *arXiv preprint arXiv:1706.10295*, 2017.

Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., and Bachem, O. Brax – A differentiable physics engine for large scale rigid body simulation. *arXiv preprint arXiv:2106.13281*, 2021.

Fujimoto, S., Hoof, H., and Meger, D. Addressing function approximation error in actor-critic methods. *International Conference on Machine Learning (ICML)*, 2018.

Geijtenbeek, T., Van De Panne, M., and Van Der Stappen, A. F. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics (TOG)*, 2013.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2010.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011.

Godfrey-Smith, P. *Other minds: The octopus, the sea, and the deep origins of consciousness*. Farrar, Straus and Giroux, 2016.

Goodfellow, I., Bengio, Y., and Courville, A. *Deep learning*. MIT press, 2016.

Google. Cloud TPU, 2018. https://cloud.google.com/tpu/.

Goyal, A., Islam, R., Strouse, D., Ahmed, Z., Botvinick, M., Larochelle, H., Bengio, Y., and Levine, S. InfoBot: Transfer and exploration via the information bottleneck. *arXiv preprint arXiv:1901.10902*, 2019.

Goyal, A., Bengio, Y., Botvinick, M., and Levine, S. The variational bandwidth bottleneck: Stochastic evaluation on an information budget. *arXiv preprint arXiv:2004.11935*, 2020.

Gregor, K., Rezende, D. J., and Wierstra, D. Variational intrinsic control. *arXiv preprint arXiv:1611.07507*, 2016.

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., and Brevdo, E. TF-Agents: A library for reinforcement learning in TensorFlow, 2018. https://github.com/tensorflow/agents.

Guez, A., Mirza, M., Gregor, K., Kabra, R., Racanière, S., Weber, T., Raposo, D., Santoro, A., Orseau, L., Eccles, T., Wayne, G., Silver, D., and Lillicrap, T. An investigation of model-free planning. *International Conference on Machine Learning (ICML)*, 2019.

Guo, J., He, H., He, T., Lausen, L., Li, M., Lin, H., Shi, X., Wang, C., Xie, J., Zha, S., Zhang, A., Zhang, H., Zhang, Z., Zhang, Z., and Zheng, S. GluonCV and GluonNLP: Deep learning in computer vision and natural language processing. *Journal of Machine Learning Research (JMLR)*, 2020.

Ha, D. and Schmidhuber, J. World models. *arXiv preprint arXiv:1803.10122*, 2018.

Haarnoja, T., Hartikainen, K., Abbeel, P., and Levine, S. Latent space policies for hierarchical reinforcement learning. *International Conference on Machine Learning (ICML)*, 2018a.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *International Conference on Machine Learning (ICML)*, 2018b.

Hafner, D., Lillicrap, T., Ba, J., and Norouzi, M. Dream to control: Learning behaviors by latent imagination. *arXiv preprint arXiv:1912.01603*, 2019a.

Hafner, D., Lillicrap, T., Fischer, I., Villegas, R., Ha, D., Lee, H., and Davidson, J. Learning latent dynamics for planning from pixels. *International Conference on Machine Learning (ICML)*, 2019b.

Hafner, D., Lillicrap, T., Norouzi, M., and Ba, J. Mastering Atari with discrete world models. *arXiv preprint arXiv:2010.02193*, 2020.

Harada, D. Reinforcement learning with time. *AAAI Conference on Artificial Intelligence*, 1997.

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., and Oliphant, T. E. Array programming with NumPy. *Nature*, 2020.

Hasenclever, L., Pardo, F., Hadsell, R., Heess, N., and Merel, J. CoMic: Complementary task learning & mimicry for reusable skills. *International Conference on Machine Learning (ICML)*, 2020.

Hausman, K., Springenberg, J. T., Wang, Z., Heess, N., and Riedmiller, M. Learning an embedding space for transferable robot skills. *International Conference on Learning Representations (ICLR)*, 2018.

Heess, N., Wayne, G., Silver, D., Lillicrap, T., Tassa, Y., and Erez, T. Learning continuous control policies by stochastic value gradients. *arXiv preprint arXiv:1510.09142*, 2015.

Heess, N., Wayne, G., Tassa, Y., Lillicrap, T., Riedmiller, M., and Silver, D. Learning and transfer of modulated locomotor controllers. *arXiv preprint arXiv:1610.05182*, 2016.

Heess, N., TB, D., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., Erez, T., Wang, Z., Eslami, S., Riedmiller, M., and Silver, D. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

Heiden, E., Millard, D., Coumans, E., Sheng, Y., and Sukhatme, G. S. NeuralSim: Augmenting differentiable simulators with neural networks. *IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. Rainbow: Combining improvements in deep reinforcement learning. *AAAI Conference on Artificial Intelligence*, 2018.

Hessel, M., Danihelka, I., Viola, F., Guez, A., Schmitt, S., Sifre, L., Weber, T., Silver, D., and Van Hasselt, H. Muesli: Combining improvements in policy optimization. *International Conference on Machine Learning (ICML)*, 2021.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 1997.

Hoffman, M., Shahriari, B., Aslanides, J., Barth-Maron, G., Behbahani, F., Norman, T., Abdolmaleki, A., Cassirer, A., Yang, F., Baumli, K., Henderson, S., Novikov, A., Gómez, S., Cabi, S., Gulcehre, C., Paine, T., Cowie, A., Wang, Z., Piot, B., and Freitas, N. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020.

Huang, S., Dossa, R., and Ye, C. CleanRL: High-quality single-file implementation of deep reinforcement learning algorithms. *GitHub repository*, 2020a. https://github.com/vwxyzjn/cleanrl.

Huang, W., Mordatch, I., and Pathak, D. One policy to control them all: Shared modular policies for agent-agnostic control. *International Conference on Machine Learning (ICML)*, 2020b.

Hutchinson, J. R., Ng-Thow-Hing, V., and Anderson, F. C. A 3D interactive method for estimating body segmental parameters in animals: Application to the turning and running performance of Tyrannosaurus rex. *Journal of Theoretical Biology*, 2007.

Hutchinson, J. R., Rankin, J. W., Rubenson, J., Rosenbluth, K. H., Siston, R. A., and Delp, S. L. Musculoskeletal modelling of an ostrich (Struthio camelus) pelvic limb: Influence of limb orientation on muscular capacity during locomotion. *PeerJ*, 2015.

Ikkala, A. and Hämäläinen, P. Converting biomechanical models from OpenSim to MuJoCo. *arXiv preprint arXiv:2006.10618*, 2020.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., and Kavukcuoglu, K. Reinforcement learning with unsupervised auxiliary tasks. *arXiv preprint arXiv:1611.05397*, 2016.

James, S., Bloesch, M., and Davison, A. J. Task-embedded control networks for few-shot imitation learning. *Conference on Robot Learning (CoRL)*, 2018.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. *ACM International Conference on Multimedia*, 2014.

Jiang, Y., Van Wouwe, T., De Groote, F., and Liu, C. K. Synthesis of biologically realistic human motion using joint torque actuation. *ACM Transactions On Graphics (TOG)*, 2019.

Jindrich, D. L., Smith, N. C., Jespers, K., and Wilson, A. M. Mechanics of cutting maneuvers by ostriches (Struthio camelus). *Journal of Experimental Biology*, 2007.

Kaelbling, L. P. Hierarchical learning in stochastic domains: Preliminary results. *International Conference on Machine Learning (ICML)*, 1993a.

Kaelbling, L. P. Learning to achieve goals. *International Joint Conference on Artificial Intelligence (IJCAI)*, 1993b.

Kapturowski, S., Ostrovski, G., Quan, J., Munos, R., and Dabney, W. Recurrent experience replay in distributed reinforcement learning. *International Conference on Learning Representations (ICLR)*, 2018.

Kearns, M. and Koller, D. Efficient reinforcement learning in factored MDPs. *International Joint Conference on Artificial Intelligence (IJCAI)*, 1999.

Kernighan, B. W. and Ritchie, D. M. *The C programming language*. Pearson Educación, 1988.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kingma, D. P. and Welling, M. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Kocsis, L. and Szepesvári, C. Bandit based Monte-Carlo planning. *European Conference on Machine Learning (ECML)*, 2006.

Kolesnikov, S. and Khrulkov, V. Sample efficient ensemble learning with Catalyst.RL. *arXiv preprint arXiv:2003.14210*, 2020.

Kossaifi, J., Panagakis, Y., Anandkumar, A., and Pantic, M. TensorLy: Tensor learning in Python. *arXiv preprint arXiv:1610.09555*, 2016.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 2012.

Kulkarni, T. D., Narasimhan, K., Saeedi, A., and Tenenbaum, J. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

La Barbera, V., Pardo, F., Tassa, Y., Daley, M., Richards, C., Kormushev, P., and Hutchinson, J. OstrichRL: A musculoskeletal ostrich simulation to study bio-mechanical locomotion. *arXiv preprint arXiv:2112.06061*, 2021.

Leary, C. and Wang, T. XLA: TensorFlow, compiled. *TensorFlow Dev Summit*, 2017.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1989.

Lee, J., Grey, M. X., Ha, S., Kunz, T., Jain, S., Ye, Y., Srinivasa, S. S., Stilman, M., and Liu, C. K. DART: Dynamic animation and robotics toolkit. *Journal of Open Source Software*, 2018a.

Lee, S., Yu, R., Park, J., Aanjaneya, M., Sifakis, E., and Lee, J. Dexterous manipulation and control with volumetric muscles. *ACM Transactions on Graphics (TOG)*, 2018b.

Lee, S., Park, M., Lee, K., and Lee, J. Scalable muscle-actuated human simulation and control. *ACM Transactions On Graphics (TOG)*, 2019.

Lenton, D., Pardo, F., Falck, F., James, S., and Clark, R. Ivy: Templated deep learning for inter-framework portability. *arXiv preprint arXiv:2102.02886*, 2021.

Li, Z., Cummings, C., and Sreenath, K. Animated Cassie: A dynamic relatable robotic character. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020.

Li, Z., Cheng, X., Peng, X. B., Abbeel, P., Levine, S., Berseth, G., and Sreenath, K. Reinforcement learning for robust parameterized locomotion control of bipedal robots. *arXiv preprint arXiv:2103.14295*, 2021.

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., and Stoica, I. RLlib: Abstractions for distributed reinforcement learning. *International Conference on Machine Learning (ICML)*, 2018.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

Lin, L.-J. *Reinforcement learning for robots using neural networks*. Carnegie Mellon University, 1992a.

Lin, L.-J. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 1992b.

Liu, L. and Hodgins, J. Learning to schedule control fragments for physics-based characters using deep Q-learning. *ACM Transactions on Graphics (TOG)*, 2017.

Liu, L. and Hodgins, J. Learning basketball dribbling skills using trajectory optimization and deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 2018.

Liu, L., Yin, K., van de Panne, M., Shao, T., and Xu, W. Sampling-based contact-rich motion control. *ACM Transactions on Graphics (TOG)*, 2010.

Lovejoy, W. S. A survey of algorithmic methods for partially observed markov decision processes. *Annals of Operations Research*, 1991.

Lynch, C., Khansari, M., Xiao, T., Kumar, V., Tompson, J., Levine, S., and Sermanet, P. Learning latent plans from play. *Conference on Robot Learning (CoRL)*, 2020.

Machado, M. C., Bellemare, M. G., Talvitie, E., Veness, J., Hausknecht, M., and Bowling, M. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *Journal of Artificial Intelligence Research (JAIR)*, 2018.

Matignon, L., Laurent, G. J., and Le Fort-Piat, N. Independent reinforcement learners in cooperative markov games: A survey regarding coordination problems. *Knowledge Engineering Review (KER)*, 2012.

Merel, J., Tassa, Y., TB, D., Srinivasan, S., Lemmon, J., Wang, Z., Wayne, G., and Heess, N. Learning human behaviors from motion capture by adversarial imitation. *arXiv preprint arXiv:1707.02201*, 2017.

Merel, J., Ahuja, A., Pham, V., Tunyasuvunakool, S., Liu, S., Tirumala, D., Heess, N., and Wayne, G. Hierarchical visuomotor control of humanoids. *arXiv preprint arXiv:1811.09656*, 2018a.

Merel, J., Hasenclever, L., Galashov, A., Ahuja, A., Pham, V., Wayne, G., Teh, Y. W., and Heess, N. Neural probabilistic motor primitives for humanoid control. *arXiv preprint arXiv:1811.11711*, 2018b.

Merel, J., Tunyasuvunakool, S., Ahuja, A., Tassa, Y., Hasenclever, L., Pham, V., Erez, T., Wayne, G., and Heess, N. Catch & carry: Reusable neural controllers for vision-guided whole-body tasks. *ACM Transactions on Graphics (TOG)*, 2020.

Metz, L., Ibarz, J., Jaitly, N., and Davidson, J. Discrete sequential prediction of continuous actions for deep RL. *arXiv preprint arXiv:1705.05035*, 2017.

Millard, M., Uchida, T., Seth, A., and Delp, S. L. Flexing computational muscle: Modeling and simulation of musculotendon dynamics. *Journal of Biomechanical Engineering*, 2013.

Minsky, M. and Papert, S. *Perceptrons*. MIT press, 1969.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

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., and Hassabis, D. Human-level control through deep reinforcement learning. *Nature*, 2015.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. *International Conference on Machine Learning (ICML)*, 2016.

Modi, V., Fulton, L., Jacobson, A., Sueda, S., and Levin, D. I. EMU: Efficient muscle simulation in deformation space. *Computer Graphics Forum*, 2021.

Moniz, J. R. A., Patra, B., and Garg, S. Compression and localization in reinforcement learning for Atari games. *arXiv preprint arXiv:1904.09489*, 2019.

Moravec, H. *Mind children: The future of robot and human intelligence*. Harvard University Press, 1988.

Munos, R., Stepleton, T., Harutyunyan, A., and Bellemare, M. G. Safe and efficient off-policy reinforcement learning. *arXiv preprint arXiv:1606.02647*, 2016.

Nachum, O., Gu, S., Lee, H., and Levine, S. Near-optimal representation learning for hierarchical reinforcement learning. *arXiv preprint arXiv:1810.01257*, 2018a.

Nachum, O., Gu, S. S., Lee, H., and Levine, S. Data-efficient hierarchical reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 2018b.

Nair, A., Pong, V., Dalal, M., Bahl, S., Lin, S., and Levine, S. Visual reinforcement learning with imagined goals. *arXiv preprint arXiv:1807.04742*, 2018.

Ng, A. Y., Harada, D., and Russell, S. Policy invariance under reward transformations: Theory and application to reward shaping. *International Conference on Machine Learning (ICML)*, 1999.

Oliphant, T. E. *A guide to NumPy*. Trelgol Publishing USA, 2006.

OpenAI. OpenAI Retro, 2018. https://github.com/openai/retro.

Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. Deep exploration via bootstrapped DQN. *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

Osband, I., Aslanides, J., and Cassirer, A. Randomized prior functions for deep reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

Ostrovski, G., Bellemare, M. G., Oord, A., and Munos, R. Count-based exploration with neural density models. *International Conference on Machine Learning (ICML)*, 2017.

Oudeyer, P.-Y., Kaplan, F., and Hafner, V. V. Intrinsic motivation systems for autonomous mental development. *IEEE Transactions on Evolutionary Computation*, 2007.

Pardo, F. Tonic: A deep reinforcement learning library for fast prototyping and benchmarking. *arXiv preprint arXiv:2011.07537*, 2020.

Pardo, F., Tavakoli, A., Levdik, V., and Kormushev, P. Time limits in reinforcement learning. *International Conference on Machine Learning (ICML)*, 2018.

Pardo, F., Levdik, V., and Kormushev, P. Scaling all-goals updates in reinforcement learning using convolutional neural networks. *AAAI Conference on Artificial Intelligence*, 2020.

Parisotto, E., Song, F., Rae, J., Pascanu, R., Gulcehre, C., Jayakumar, S., Jaderberg, M., Kaufman, R. L., Clark, A., Noury, S., Botvinick, M. M., Heess, N., and Hadsell, R. Stabilizing transformers for reinforcement learning. *International Conference on Machine Learning (ICML)*, 2020.

Parr, R. and Russell, S. Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems (NeurIPS)*, 1997.

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., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. *International Conference on Machine Learning (ICML)*, 2017.

Pavlov, M., Kolesnikov, S., and Plis, S. Run, skeleton, run: Skeletal model in a physics-based simulation. *AAAI Spring Symposium Series*, 2018.

Pawlowski, N., Ktena, S. I., Lee, M. C., Kainz, B., Rueckert, D., Glocker, B., and Rajchl, M. DLTK: State of the art reference implementations for deep learning on medical images. *arXiv preprint arXiv:1711.06853*, 2017.

Peng, X. B., Berseth, G., Yin, K., and Van De Panne, M. DeepLoco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 2017.

Peng, X. B., Abbeel, P., Levine, S., and van de Panne, M. DeepMimic: Example-guided deep reinforcement learning of physics-based character skills. *ACM Transactions on Graphics (TOG)*, 2018.

Peng, X. B., Chang, M., Zhang, G., Abbeel, P., and Levine, S. MCP: Learning composable hierarchical control with multiplicative compositional policies. *arXiv preprint arXiv:1905.09808*, 2019.

Pierrot, T., Macé, V., Cideron, G., Beguir, K., Cully, A., Sigaud, O., and Perrin-Gilbert, N. Diversity policy gradient for sample efficient quality-diversity optimization. *arXiv preprint arXiv:2006.08505*, 2020.

Plappert, M. Keras-RL, 2016. https://github.com/keras-rl/keras-rl.

Plappert, M., Houthooft, R., Dhariwal, P., Sidor, S., Chen, R. Y., Chen, X., Asfour, T., Abbeel, P., and Andrychowicz, M. Parameter space noise for exploration. *arXiv preprint arXiv:1706.01905*, 2017.

Potvin, J. R. and Fuglevand, A. J. A motor unit-based model of muscle fatigue. *PLOS Computational Biology*, 2017.

Precup, D. *Temporal abstraction in reinforcement learning*. University of Massachusetts Amherst, 2000.

Precup, D., Sutton, R. S., and Singh, S. Theoretical results on reinforcement learning with temporally abstract options. *European Conference on Machine Learning (ECML)*, 1998.

Pritzel, A., Uria, B., Srinivasan, S., Badia, A. P., Vinyals, O., Hassabis, D., Wierstra, D., and Blundell, C. Neural episodic control. *International Conference on Machine Learning (ICML)*, 2017.

Péré, A., Forestier, S., Sigaud, O., and Oudeyer, P.-Y. Unsupervised learning of goal spaces for intrinsically motivated goal exploration. *arXiv preprint arXiv:1803.00781*, 2018.

Qiao, Y.-L., Liang, J., Koltun, V., and Lin, M. C. Scalable differentiable physics for learning and control. *arXiv preprint arXiv:2007.02168*, 2020.

Racanière, S., Weber, T., Reichert, D. P., Buesing, L., Guez, A., Rezende, D., Badia, A. P., Vinyals, O., Heess, N., Li, Y., Pascanu, R., Battaglia, P., Hassabis, D., Silver, D., and Wierstra, D. Imagination-augmented agents for deep reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Raissi, M., Yazdani, A., and Karniadakis, G. E. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. *Science*, 2020.

Rankin, J. W., Rubenson, J., and Hutchinson, J. R. Inferring muscle functional roles of the ostrich pelvic limb during walking and running using computer optimization. *Journal of the Royal Society Interface*, 2016.

Ratliff, N., Zucker, M., Bagnell, J. A., and Srinivasa, S. CHOMP: Gradient optimization techniques for efficient motion planning. *IEEE International Conference on Robotics and Automation (ICRA)*, 2009.

Ravi, N., Reizenstein, J., Novotny, D., Gordon, T., Lo, W.-Y., Johnson, J., and Gkioxari, G. PyTorch3D, 2020. https://github.com/facebookresearch/pytorch3d.

Reher, J., Ma, W.-L., and Ames, A. D. Dynamic walking with compliance on a Cassie bipedal robot. *European Control Conference (ECC)*, 2019.

Riedmiller, M., Hafner, R., Lampe, T., Neunert, M., Degrave, J., Wiele, T., Mnih, V., Heess, N., and Springenberg, J. T. Learning by playing solving sparse reward tasks from scratch. *International Conference on Machine Learning (ICML)*, 2018.

Rummery, G. A. and Niranjan, M. *On-line Q-learning using connectionist systems*. Citeseer, 1994.

Sawada, Y. Disentangling controllable and uncontrollable factors of variation by interacting with the world. *arXiv preprint arXiv:1804.06955*, 2018.

Schaul, T., Horgan, D., Gregor, K., and Silver, D. Universal value function approximators. *International Conference on Machine Learning (ICML)*, 2015a.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015b.

Schmidhuber, J. A possibility for implementing curiosity and boredom in model-building neural controllers. *International Conference on Simulation of Adaptive Behavior: From Animals to Animats*, 1991.

Schrader, M.-P. B. Gym-Sokoban, 2018. https://github.com/mpSchrader/gym-sokoban.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., Lillicrap, T., and Silver, D. Mastering Atari, Go, Chess and Shogi by planning with a learned model. *Nature*, 2020.

Schulman, J., Duan, Y., Ho, J., Lee, A., Awwal, I., Bradlow, H., Pan, J., Patil, S., Goldberg, K., and Abbeel, P. Motion planning with sequential convex optimization and convex collision checking. *International Journal of Robotics Research (IJRR)*, 2014.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., and Moritz, P. Trust region policy optimization. *International Conference on Machine Learning (ICML)*, 2015a.

Schulman, J., Moritz, P., Levine, S., Jordan, M., and Abbeel, P. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015b.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Seide, F. and Agarwal, A. CNTK: Microsoft's open-source deep-learning toolkit. *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2016.

Sellers, W., Margetts, L., Bates, K., and Chamberlain, A. Exploring diagonal gait using a forward dynamic three-dimensional chimpanzee simulation. *Folia Primatologica*, 2013.

Seth, A., Hicks, J., Uchida, T., Habib, A., Dembia, C., Dunne, J., Ong, C., DeMers, M., Rajagopal, A., Millard, M., Hamner, S., Arnold, E., Yong, J., Lakshmikanth, S., Sherman, M., and Delp, S. OpenSim: Simulating musculoskeletal dynamics and neuromuscular control to study human and animal movement. *PLOS Computational Biology*, 2018.

Shi, J., Chen, J., Zhu, J., Sun, S., Luo, Y., Gu, Y., and Zhou, Y. ZhuSuan: A library for bayesian deep learning. *arXiv preprint arXiv:1709.05870*, 2017.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. Deterministic policy gradient algorithms. *International Conference on Machine Learning (ICML)*, 2014.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. Mastering the game of Go with deep neural networks and tree search. *Nature*, 2016.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. Mastering Chess and Shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017a.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. Mastering the game of Go without human knowledge. *Nature*, 2017b.

Smith, M., Hoof, H., and Pineau, J. An inference-based policy gradient method for learning options. *International Conference on Machine Learning (ICML)*, 2018.

Smith, R. et al. Open dynamics engine, 2005.

Song, H. F., Abdolmaleki, A., Springenberg, J., Clark, A., Soyer, H., Rae, J., Noury, S., Ahuja, A., Liu, S., Tirumala, D., Heess, N., Belov, D., Riedmiller, M., and Botvinick, M. V-MPO: On-policy maximum a posteriori policy optimization for discrete and continuous control. *arXiv preprint arXiv:1909.12238*, 2019.

Stadie, B. C., Levine, S., and Abbeel, P. Incentivizing exploration in reinforcement learning with deep predictive models. *arXiv preprint arXiv:1507.00814*, 2015.

Stark, H., Fischer, M. S., Hunt, A., Young, F., Quinn, R., and Andrada, E. A three-dimensional musculoskeletal model of the dog. *Scientific Reports*, 2021.

Stolle, M. and Precup, D. Learning options in reinforcement learning. *International Symposium on Abstraction, Reformulation, and Approximation*, 2002.

Stooke, A. and Abbeel, P. rlpyt: A research code base for deep reinforcement learning in PyTorch. *arXiv preprint arXiv:1909.01500*, 2019.

Stroustrup, B. *The C++ programming language*. Pearson Education India, 2000.

Sutton, R. S. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Machine Learning Proceedings*, 1990.

Sutton, R. S. Planning by incremental dynamic programming. *Machine Learning Proceedings*, 1991.

Sutton, R. S. TD models: Modeling the world at a mixture of time scales. *Machine Learning Proceedings*, 1995.

Sutton, R. S. and Barto, A. G. *Reinforcement learning: An introduction*. MIT press, 2018.

Sutton, R. S., Precup, D., and Singh, S. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 1999.

Sutton, R. S., McAllester, D. A., Singh, S. P., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems (NeurIPS)*, 2000.

Sutton, R. S., Modayil, J., Delp, M., Degris, T., Pilarski, P. M., White, A., and Precup, D. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2011.

Szepesvári, C. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 2010.

Tamar, A., Wu, Y., Thomas, G., Levine, S., and Abbeel, P. Value iteration networks. *arXiv preprint arXiv:1602.02867*, 2016.

Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R. Multiagent cooperation and competition with deep reinforcement learning. *PLOS One*, 2017.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, X., Duan, Y., Schulman, J., De Turck, F., and Abbeel, P. #Exploration: A study of count-based exploration for deep reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Tassa, Y., Doron, Y., Muldal, A., Erez, T., Li, Y., Casas, D. d. L., Budden, D., Abdolmaleki, A., Merel, J., Lefrancq, A., Lillicrap, T., and Riedmiller, M. DeepMind control suite. *arXiv preprint arXiv:1801.00690*, 2018.

Tassa, Y., Tunyasuvunakool, S., Muldal, A., Doron, Y., Liu, S., Bohez, S., Merel, J., Erez, T., Lillicrap, T., and Heess, N. dm_control: Software and tasks for continuous control. *arXiv preprint arXiv:2006.12983*, 2020.

Tavakoli, A., Pardo, F., and Kormushev, P. Action branching architectures for deep reinforcement learning. *AAAI Conference on Artificial Intelligence*, 2018.

Tesauro, G. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 1994.

Tieleman, T. and Hinton, G. Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 2012.

Todorov, E., Erez, T., and Tassa, Y. MuJoCo: A physics engine for model-based control. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2012.

Tokui, S., Okuta, R., Akiba, T., Niitani, Y., Ogawa, T., Saito, S., Suzuki, S., Uenishi, K., Vogel, B., and Yamazaki Vincent, H. Chainer: A deep learning framework for accelerating the research cycle. *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, 2019.

Tsuihiji, T. Homologies of the transversospinalis muscles in the anterior presacral region of Sauria (crown Diapsida). *Journal of Morphology*, 2005.

Tsuihiji, T. Homologies of the longissimus, iliocostalis, and hypaxial muscles in the anterior presacral region of extant diapsida. *Journal of Morphology*, 2007.

Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *London Mathematical Society*, 1936.

Turing, A. M. Computing machinery and intelligence. *Mind*, 1950.

Uhlenbeck, G. E. and Ornstein, L. S. On the theory of the Brownian motion. *Physical Review*, 1930.

Valentin, J., Keskin, C., Pidlypenskyi, P., Makadia, A., Sud, A., and Bouaziz, S. TensorFlow graphics: Computer graphics meets deep learning. *TensorFlow Graphics IO*, 2019.

Van, R. G. and Drake, F. Python 3 reference manual. *CreateSpace*, 2009.

Van de Wiele, T., Warde-Farley, D., Mnih, A., and Mnih, V. Q-learning in enormous action spaces via amortized approximate maximization. *arXiv preprint arXiv:2001.08116*, 2020.

Van Hasselt, H. Double Q-learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 2010.

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double Q-learning. *AAAI Conference on Artificial Intelligence*, 2016.

Van Seijen, H., Fatemi, M., Romoff, J., Laroche, R., Barnes, T., and Tsang, J. Hybrid reward architecture for reinforcement learning. *arXiv preprint arXiv:1706.04208*, 2017.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Veeriah, V., Oh, J., and Singh, S. Many-goals reinforcement learning. *arXiv preprint arXiv:1806.09605*, 2018.

Vezhnevets, A., Mnih, V., Agapiou, J., Osindero, S., Graves, A., Vinyals, O., and Kavukcuoglu, K. Strategic attentive writer for learning macro-actions. *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. Feudal networks for hierarchical reinforcement learning. *International Conference on Machine Learning (ICML)*, 2017.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., Choi, D. H., Powell, R., Ewalds, T., Georgiev, P., Oh, J., Horgan, D., Kroiss, M., Danihelka, I., Huang, A., Sifre, L., Cai, T., Agapiou, J. P., Jaderberg, M., Vezhnevets, A. S., Leblond, R., Pohlen, T., Dalibard, V., Budden, D., Sulsky, Y., Molloy, J., Paine, T. L., Gulcehre, C., Wang, Z., Pfaff, T., Wu, Y., Ring, R., Yogatama, D., Wünsch, D., McKinney, K., Smith, O., Schaul, T., Lillicrap, T., Kavukcuoglu, K., Hassabis, D., Apps, C., and Silver, D. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature*, 2019.

Wang, J., Gao, D., and Lee, P. S. Recent progress in artificial muscles for interactive soft robotics. *Advanced Materials*, 2021.

Wang, J. M., Hamner, S. R., Delp, S. L., and Koltun, V. Optimizing locomotion controllers using biologically-based actuators and objectives. *ACM Transactions on Graphics (TOG)*, 2012.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., and Freitas, N. Dueling network architectures for deep reinforcement learning. *International Conference on Machine Learning (ICML)*, 2016.

Wang, Z., Merel, J., Reed, S., Wayne, G., de Freitas, N., and Heess, N. Robust imitation of diverse behaviors. *arXiv preprint arXiv:1707.02747*, 2017.

Warde-Farley, D., Van de Wiele, T., Kulkarni, T., Ionescu, C., Hansen, S., and Mnih, V. Unsupervised control through non-parametric discriminative rewards. *arXiv preprint arXiv:1811.11359*, 2018.

Watkins, C. J. and Dayan, P. Q-learning. *Machine Learning*, 1992.

White, M. Unifying task specification in reinforcement learning. *International Conference on Machine Learning (ICML)*, 2017.

Whitehead, S. D. and Ballard, D. H. Learning to perceive and act by trial and error. *Machine Learning*, 1991.

Wiering, M. and Schmidhuber, J. HQ-learning. *Adaptive Behavior*, 1997.

Wierstra, D., Förster, A., Peters, J., and Schmidhuber, J. Recurrent policy gradients. *Logic Journal of the IGPL*, 2010.

Williams, R. J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.

Won, J., Gopinath, D., and Hodgins, J. Control strategies for physically simulated characters performing two-player competitive sports. *ACM Transactions on Graphics (TOG)*, 2021.

Wulfmeier, M., Abdolmaleki, A., Hafner, R., Springenberg, J. T., Neunert, M., Hertweck, T., Lampe, T., Siegel, N., Heess, N., and Riedmiller, M. Compositional transfer in hierarchical reinforcement learning. *arXiv preprint arXiv:1906.11228*, 2019.

Xie, Z., Berseth, G., Clary, P., Hurst, J., and van de Panne, M. Feedback control for Cassie with deep reinforcement learning. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

Xie, Z., Clary, P., Dao, J., Morais, P., Hurst, J., and Panne, M. Learning locomotion skills for Cassie: Iterative design and sim-to-real. *Conference on Robot Learning (CoRL)*, 2020.

Yin, Z., Yang, Z., Van De Panne, M., and Yin, K. Discovering diverse athletic jumping strategies. *ACM Transactions on Graphics (TOG)*, 2021.

Zajac, F. E. Muscle and tendon: Properties, models, scaling, and application to biomechanics and motor control. *Critical Reviews in Biomedical Engineering (CRB)*, 1989.

Zhang, S. and Sutton, R. S. A deeper look at experience replay. *arXiv preprint arXiv:1712.01275*, 2017.

Zhou, B., Zeng, H., Wang, F., Li, Y., and Tian, H. Efficient and robust reinforcement learning with uncertainty-based value expansion. *arXiv preprint arXiv:1912.05328*, 2019.