THE UNIVERSITY OF EDINBURGH

HONOURS PROJECT

Exploring Bipedal Walking Through Machine Learning Techniques

Author:
Levi FUSSELL

Supervisor:
Dr. Taku KOMURA

BSc. Computer Science and Artificial Intelligence

April 12, 2018

THE UNIVERSITY OF EDINBURGH

Abstract

Exploring Bipedal Walking Through Machine Learning Techniques

by Levi FUSSELL

This project is about an agent learning to walk. But, from this general goal we accomplished many things and encountered many insights. The agent was taught how to walk via two unique learning algorithms: *Deep Deterministic Policy Gradients (DDPG)* and *Evolutionary Strategies (ES)*. During this process we explored the central aspects of each algorithm independently, and then also compared the algorithms using the maximum *mean return*. We found that although *ES* achieved the highest *mean return*, *DDPG* displayed the more realistic walking behaviour. By developing three different evaluation metrics for valuing the 'human likeness' of a learned gait, we found *neither* algorithm was walking optimally.

In the second half of this project, we explored methods for producing more human-like gaits. The *ES* algorithm was trained with two types of recurrent neural networks to attempt to improve the 'richness' of its gait. This failed and made *ES* perform *worse*. The *DDPG* algorithm was trained with three novel multi-controller architectures: a two-output symmetrical controller (TOS), a hierarchical controller (HC), and a distributed controller (DC). We claimed success with the HC and DC model, as they both proved complex walking gaits *can* be learned from simple reward functions. The conclusion provides a thorough comparison of *all* the models we tested, and some closing remarks concerning how the our novel control models could be extended to different problems other than walking.

Acknowledgements

I would like to acknowledge the following people who have helped me complete this project:

My supervisor *Dr. Taku Komura*, who provided valuable insight and guidance on the project and final report. In addition, he answered any and all of my questions.

My friend *Thais Minet*, who read a rough draft of this report, cover to cover, and meticulously pointed out many errors.

Contents

List of Figures

List of Tables

List of Abbreviations

DDPG	Deep Deterministic Policy Gradients
ES	Evolutionary Strategies
CE	Cross Entropy (Method)
PGM	Policy Gradient Method
MLP	Multi-Layer Perceptron
FFNN	Feed-Forward Neural Network
RNN	Recurrent Neural Network
SGD	Stochastic Gradient Descent
i.i.d.	independently and identically distributed
RAM	Random Access Memory
JN	Jordan Network
FR	Fully-Recurrent (Network)
TOS	Two Output Symmetrical (Model)
HC	Hierarchical Controller (Model)
DC	Distributed Controller (Model)
NS	Node Scale
POP	Population

Chapter 1

Introduction

Despite the valiant efforts of history's grand masters and professionals, the games Chess [67], Go [62], Atari [45], and even Catan [51] are succumbing - much sooner than many had anticipated - to the raw power of computing and reinforcement learning. A different problem domain is the set of problems involving continuous-action dynamics for controlling robots in the real world. These types of problems have not been 'solved' and any human is more superior at these tasks than the best robot has ever been. The difficulty of controlling robots comes from the infinite number of possible actions and possible states the robot can be in [24], making exploration and generalisation impossible if the deterministic techniques that solved the likes of Chess and Go are used. Despite many challenges, progress on these continuous-action problems has been incredible in the past few decades.

A recent paper by OpenAI showed how two polar approaches to solving continuous-action problems, *evolutionary strategies* and *policy gradient methods*, both achieve comparable performance on various non-trivial environments if hardware constraints are excluded [60]. The metric that was used in the paper to evaluate the performance of the two algorithms was the maximum *mean return* over time (see chapter 2), but no experiments were done to determine both the realism and generalisability of the solutions that the two algorithms produced. Focusing solely on the *mean return* can lead to problems such as over-fitting: where a system becomes so adept in one domain that it is unable to perform well in a new, unseen domain [81].

In this report our *main goal* was the following:

> *Explore whether optimum policies learned by machine learning methods equate to 'natural' and useful policies found in real life, in the context of learning to walk.*

We divided this into two problems. Firstly, we wished to develop an initial understanding of the two algorithms, *evolutionary strategies* and *policy gradient methods*, and do a comparative analysis of them. Following this, we then wanted to see how well each of the algorithms learned a *robust* and *realistic* solution to a complex continuous-action control problem. For the sake of rigor and time, we scoped our experiments to the problem of walking on two legs - referred to as a *bipedal walk* - in a two-dimensional environment. Initially, we used the maximum *mean return*, as is done in the OpenAI paper [60], as a heuristic to evaluate the algorithms and filter out unanimously bad models. We showed how using the maximum *mean return* does *not* imply a human-like gait [1] has been learned. We then introduced some new methods to determine if the walk that had been learned was robust and human-like.

[1] The terms *'human-like'* and *'natural'* will be used incessantly in this report to refer to a gait that qualitatively is similar to how humans walk. There really is no formal way to refer to this.

The first chapter provides background on the material covered in this report. The chapter is divided across two domains: the field of reinforcement learning and the field of walking dynamics. Specifically, Section 2.1 outlines the reinforcement learning problem in a general context and introduces the different algorithms we will use to solve the walking environment. In Section 2.2 we introduce the neural network and how it applies to the field of deep reinforcement learning. We then switch to the walking domain in Section 2.3 and provide a brief overview of some interesting physics and biology behind a human bipedal walk; this inspired our novel architectures in Chapter 6. Finally, in Chapter 3 we present the details of the environment and the walking agent we worked with, as well as introduce the methodology we used for evaluating the walks of the agent.

The next part of this report includes experimental results and discussions of these results, and an attempt to go beyond the original implementations to try to produce a more natural gait using hierarchical and distributed model architectures. In Chapter 4 we start with a simple agent-balancing environment to unit test the implementations of our algorithms. Also in Chapter 4, we present an analysis of the algorithm's capacity to solve the walking environment in the context of the maximum *mean return*. We then evaluate the algorithms on the basis of how human-like the gait is, in Chapter 5. Then in Chapter 6 we address the failures of the models to learn a human-like gait and attempt to improve the gait by changing the model. Lastly, an comparison of all models in this report is provided in Chapter 7, and some conclusive remarks and a write-up on future work is provided in Chapter 8.

1.1 Goals

Achieving the main goal 1 of this report required designing and completing many supplementary goals. We outline the subgoals we defined in the following list. A reflection on these goals is provided at the end of Chapter 5.

1. Learn about deep reinforcement learning as a precursor to implementation.
2. Implement successful walking behaviours in the *Bipedal Walker v2 OpenAI Gym* environment using deep reinforcement learning, evolutionary strategies, and a baseline method.
3. Evaluate the results on the basis of the maximum *mean return* over time.
4. Develop metrics to *quantitatively* analyse a human-like bipedal gait.
5. Based on the metrics developed earlier, research and explore alterations to the model that could improve the realism of the bipedal gait.

1.2 Work Completed & Contributions

In addition to the defined subgoals, there was specific *work* that we had to do in order to reach these goals. This work is the total sum of what we have accomplished to reach the main goal and defines the time investment required for completing this project. In addition, we outline our *contributions* which is defined as the work we have contributed to the research community.

Work Completed

1. Implemented first REINFORCE and neural network model for initial tests in Python (based on Karparthy implementation[2]).
2. Built a framework in PyTorch for various Policy Gradient algorithms (REINFORCE, REINFORCE plus baseline, DDPG).
3. Built a framework in Python for running Evolutionary Strategies. This framework was faster than our initial testing framework (Evostra [3]).
4. Built a framework in Python for running Cross Entropy Methods.
5. Implemented the Bipedal Balancer Environment from the OpenAI Gym framework.
6. Devised and implemented various methods to quantitatively evaluate a bipedal walk based on how 'natural' it is.

Contributions

1. Developed two novel architectures for improving the quality of the bipedal gait. One is based on the hierarchical control relationship between the brain and the nervous system. The second is a distributed control architecture.
2. Because the architectures require unique training processes, we plan to publish the source code for them after the completion of this project.

[2]Found here: https://gist.github.com/karpathy/a4166c7fe253700972fcbc77e4ea32c5

Chapter 2

Background

This chapter provides preliminary background for the experiments and topics we discuss later in the report. The chapter is divided into three separate sections: reinforcement learning, neural networks, and bipedal walking. The reinforcement learning section introduces *what* reinforcement learning is, and then defines the theory of the three algorithms we will be using in this report. The neural networks section is self contained because it introduces the model we use to control the agents, independent of the algorithm. We also provide insight on how neural networks combine with reinforcement learning in the field of deep reinforcement learning. The bipedal walking section helps familiarise the reader with the mechanics and biology behind a human gait.

2.1 Reinforcement Learning

Reinforcement learning (RL) is learning how to behave and act purely through interactions with an environment and through numerical reward signals indicating how your actions contributed to (or hindered) progress towards a goal [72] [23]. It unifies concepts from control theory, machine learning, and psychology to try to produce generalised learning. The consensus among some psychologist is that this simplified analogy of reward feedback is strongly related to how the brain works [46] [80]. The advent of this theory began with the *anhedonia hypothesis* concerning the use of dopamine as a positive-reward feedback in the brain [84]. Since then, reinforcement learning as a field has blossomed from a theoretical framework to a mathematical construct where these theories can be implemented and tested in computer simulations or real robots.

We now proceed to define reinforcement learning in a formal domain. The field of reinforcement learning is vast, so we chose to focus on the minimum amount required to understand the algorithms used in this report. We discuss how to construct a reinforcement learning problem for some task, the definitions of reward, return, policy, and value function, and, lastly, the exploration vs. exploitation problem and the difference between on-policy and off-policy learning. For further insight beyond the material covered, we point the reader to Sutton and Barto's book, *Reinforcement Learning: An Introduction* [72].

2.1.1 The Reinforcement Learning Problem

A reinforcement learning (RL) problem is described by the interaction between an agent and its environment through states, actions, and reward signals [72].

An *environment* is some system, in our case a two-dimensional world with physical interactions, in which an agent perceives and interacts.

The environment and its agent move through time together in single units of time (chronons [12]). At each time step the environment E tells the agent information about its state $\mathbf{s}$ at time step t, via an observation $\mathbf{o}$, and the agent performs an action $\mathbf{a}$ on the environment receiving a reward r. Then the cycle repeats, as shown in Figure 2.1.

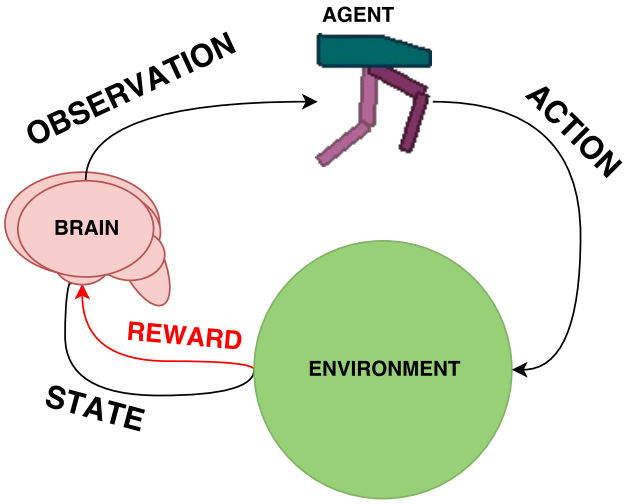


FIGURE 2.1: Diagram depicting the interaction between the agent and its environment for an RL problem.

The environment and the agent's interactions are modeled by a Markov Decision Process (MDP) which is defined by:

1. A set of *states* S the environment can be in. A state $\mathbf{s} \in \mathbb{R}^N$ is a vector of information about the environment if the environment were frozen at time t.
2. A set of *actions* A describing the possible movements of the agent. An action $\mathbf{a} \in \mathbb{R}^N$ is a vector describing what the agent chooses to do in the environment at that point in time. For example $\mathbf{a} = (0.5, 1.0, 0.1, 0.0)$ might describe torque values to apply to an agent's four joints
3. *Transition probabilities* $P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) : S \times S \times A \to [0, 1]$ representing the probability of the environment being in state $\mathbf{s}_{t+1}$ given the agent and the environment are currently in state $\mathbf{s}_t$ and the agent has taken action $\mathbf{a}_t$. States are connected via actions and transition probabilities. Given two states $\mathbf{s}_1, \mathbf{s}_2 \in S$ and some action $\mathbf{a}_1 \in A$ which an agent has taken while in state $\mathbf{s}_1$, the probability that the agent will end up in the state $\mathbf{s}_2$ is $P(\mathbf{s}_2|\mathbf{s}_1, \mathbf{a}_1)$. The transition probabilities therefore model stochasticity in the environment. Throwing a paper ball when it is windy outside is stochastic as you can't predict exactly where it will land.

We represent these properties of an MDP by the tuple (S, A, P). An MDP gets its name from the fact it uses a strong, yet useful, assumption called the *Markov Property*:

Definition 2.1.1 (Markov Property) *Given some current state of a system* $\mathbf{s}_t$*, an action* $\mathbf{a}_t$*, and a subsequent state* $\mathbf{s}_{t+1}$ *resulting from taking* $\mathbf{a}_t$ *while the system is in state* $\mathbf{s}_t$*, then it holds that some transition function* $P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) : S \times S \times A \to [0, 1]$ *has the Markov Property if:*

$$P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) = P(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t-1}, \mathbf{a}_{t-1}, .., .., \mathbf{s}_0, \mathbf{a}_0)$$

In other words, events occurring at time $t + 1$ are only dependent on events that occurred immediately before at time t and we ignore dependence on *all* previous events until the start of time.

The *Markov Reward Process* (MRP) extends the MDP by incorporating a reward function, $R : S \to \mathbb{R}$ which returns a reward value given the state $\mathbf{s}$ of the environment [72]. An MDP is then a *Partially Observable MDP* (POMDP) if there is information about the environment which the agent cannot know or which is obscure [72]. For example, in the environment of the card game Poker, the hands of the other players are not known to a poker-playing agent, therefore Poker is a POMDP. If a POMDP incorporates a reward function *as well*, then it is a *Partially Observable MRP* (POMRP). We represent this partial-observability by a set of observations $\mathbf{o} \in O$ describing the information that the agent receives from the environment. It is not necessarily true that $O \subseteq S$ because the observations may contain redundant information. The walking environment we explored is partially observable because the agent does not have complete information about factors such as the terrain it is walking on. Because our environment also contains a reward system, it is completely defined by a POMRP (S, O, A, P, R).

Once an environment has been defined in terms of some MDP, or variant of, it becomes more natural to solve the reinforcement learning problem for the general task. We now elaborate on the definitions such as *policy*, *return*, and *action-value* used to further enrich the description of the reinforcement learning problem.

Rewards and Returns

As described earlier, for each action the agent performs there is an associated reward. Reinforcement learning problems share a common goal, which is to maximise the amount of reward an agent receives over time. Formally, we define the expected reward over time as the *return*: $G_t = \sum_{i=1}^{\inf} \gamma^{i-1} R_{t+i}$ [72].

This equation tells us how much reward an agent will receive starting from some state at time step t and acting into the infinite future, where future rewards are discounted by γ^i, i being the number of time steps into the future the reward is and $\gamma \in [0, 1]$. $\gamma \approx 0$ indicates that only immediate rewards should be considered and $\gamma \approx 1$ gives equal weighting to all rewards in the future. Intuitively, this reward discounting makes sense at a human level. If you were offered one thousand GBP in one year or one million GBP in one-hundred years, the closer reward, although a smaller quantity, is more useful than the future reward because you may not be alive in one-hundred years [63]. Using the future return instead of the immediate reward is generally more insightful as it provides information about how good the agent's state will be in the *future* if the agent takes some action. A sequence of rewards $+1, +1, +1, +1, +1, +1, ...$ is better than a sequence of rewards

$+5, -1, -1, -1, -1, -1, ...$ even though the immediate reward, $+1$ is less than $+5$. The *return* gives the agent the ability of *insight*.

An active area of interest in reinforcement learning is how to teach an agent to use *sparse rewards* [4] [6]. Sparse rewards are in environments where the majority of the states have a reward value of zero, with only a few states having non-zero reward values. As most situations in reality contain sparse rewards (i.e. monthly paychecks or winning a game of chess), this is a relevant problem to try to solve. The problem with sparse rewards is how an agent allocates the reward to the actions that were taken leading up to the reward [68]. Consider the imaginary environment based on a game of tennis where an agent receives a +1 reward if it wins the match and a -1 reward if it loses the match. Winning a match requires playing multiple sets, each of which requires winning multiple games, which require winning multiple points. In this environment, it is difficult for an agent to distinguish good and bad actions, as the good and bad action can result in the same reward. A reward function that would allow an agent to learn easier would be one that reinforces subtasks. For example, the reward function could have a small positive reward for the agent returning the tennis ball over the net.

Policy

The *policy* π is a function that tells the agent its next action, given the agent's current state. It corresponds to stimulus-response associations commonly found in biological organisms [72]. Formally, a policy is a deterministic function $\pi : S \rightarrow A$, which maps the agent's actions to states such that $\mathbf{a}_t = \pi(\mathbf{s}_t)$. A policy can also be stochastic such that it outputs a probability distribution over the possible actions given an input of the current state $\pi : S \times A \rightarrow [0, 1]$; specifically, $P(\mathbf{a}_t|\mathbf{s}_t) = \pi(\mathbf{a}_t|\mathbf{s}_t)$. Both are useful for certain problems. For example, a deterministic policy might get stuck in suboptimal loops in the environment. As the policy is just a function, it is a natural question to wonder if we can model this function. A neural network is a universal function approximator [15] [14] and is therefore a good candidate; this is discussed in further detail in Section 2.2.

Value Functions

In order for an agent to learn a policy that produces high reward, the agent must be able to value its policy relative to other policies it could have chosen. We can value a policy through either the states the policy takes us to, or the actions the policy performs. We use the *value function* to represent this valuation by associating the state the agent is in or an action the agent took under some policy with the expected return.

If an agent is in state $\mathbf{s}_t$ and performs action $\mathbf{a}_t$ we can define a function $Q : S \times A \rightarrow \mathbb{R}$ that outputs a real value which is the expected return. Formally, the *Bellman Expectation Equation* [8] describes this relationship:

$$Q(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}[G|\mathbf{s}_t, \mathbf{a}_t] = r_{t+1} + \gamma \sum_{\mathbf{s}_{t+1} \in S} P(\mathbf{s}_t|\mathbf{s}_{t+1}, \mathbf{a}_t) V(\mathbf{s}_{t+1})$$

where r_{t+1} is the reward gained for taking the action $\mathbf{a}_t$ and $V(\mathbf{s}_{t+1})$ is notation for the expected return received from the new state $\mathbf{s}_{t+1}$ into the future. (sidenote: $V(\mathbf{s}_{t+1})$ should not be confused with the actual return G_{t+1}) Intuitively, $V(\mathbf{s}_{t+1})$ can be defined as a function $V : S \to \mathbb{R}$ and can be evaluated from a stochastic policy such that:

$$V(\mathbf{s}_t) = \sum_{\mathbf{a}_t \in A} \pi(\mathbf{a}_t|\mathbf{s}_t) Q(\mathbf{s}_t, \mathbf{a}_t)$$

By combining the two equations we get:

$$Q(\mathbf{s}_t, \mathbf{a}_t) = r_{t+1} + \gamma \sum_{\mathbf{s}_{t+1} \in S} P(\mathbf{s}_t|\mathbf{s}_{t+1}, \mathbf{a}_t) \sum_{\mathbf{a}_{t+1} \in A} \pi(\mathbf{a}_t|\mathbf{s}_t) Q(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}))$$

This 'Q-function' is interesting because if we can calculate such a function, we can 'bootstrap' the function recursively to compute the expected return of the agent in a single step, instead of having to run an entire episode to determine the return.

We define an optimal policy π^*as a policy in which $Q^*(\mathbf{s}, \mathbf{a}) \geq Q(\mathbf{s}, \mathbf{a})$ for all $\mathbf{s} \in S, \mathbf{a} \in A$ and for any policy π. This is the policy that, if our agent were to take, would maximise the agent's return better than any other policy. The *Bellman Optimality Equation* [8][64] states that the optimal Q-function can be computed (and hence the optimal policy can be found) by choosing actions that maximise the Q-function:

$$Q^*(\mathbf{s}_t, \mathbf{a}_t) = r_{t+1} + \gamma \sum_{\mathbf{s}_{t+1} \in S} P(\mathbf{s}_t|\mathbf{s}_{t+1}, \mathbf{a}_t) max_{\mathbf{a} \in A} \pi(\mathbf{a}_{t+1}|\mathbf{s}_{t+1}) Q^*(\mathbf{s}_{t+1}, \mathbf{a}_{t+1})$$

The optimal policy is then:

$$\pi^*(s) = argmax_{\mathbf{a}_\in A} Q^*(\mathbf{s}_t, \mathbf{a}_t)$$

The maths translates to the intuition: an agent should take the action which will give it the most return. If an agent does this for every state, then it will maximise its possible return.

Exploration vs. Exploitation

If an agent always follows its best policy, it may never encounter new states where its policy may fail, or new actions that could improve its policy. Alternatively, if the agent is always exploring new states and actions, it will never exploit its best policy to maximise its return. Deciding whether an agent should perform the first or second option is the problem of *exploration versus exploitation*. Despite the proof that there exists an optimal policy, π^*, for some MDP, we are not guaranteed convergence in a reasonable time [64]. Additionally, for a *model-free* problem (that is, an environment where the agent has no prior information about the states or dynamics

between states), the agent must discover states of the environment in order to find the optimal policy.

It is popular to introduce this concept in the context of the multi-armed bandit problem [5] [72]:

> You enter a casino room with eight slot machines (a single slot machine is colloquially referred to as a 'one-armed bandit') and the intent to get rich in one hour. Each machine will output different amounts of money, some more often than others and some with larger quantities. The problem is, you can only use one machine at a time and therefore cannot know the 'quality' of a machine's output until you start using it a few times (or a hundred times..). How do you make the most money? Do you stick to the first machine that outputs the most money (exploitation)? Or do you explore each machine trying to perfectly model the amount each machine will give you, but wasting precious time (exploration)?

In reinforcement learning, there is a strong but useful assumption that if the agent is using a policy, π_0, then there exists some other policy, π_1, such that $\pi_1 > \pi_0$, and this policy is most likely a slight deviation from π_0. If policy π_0 has a sequence of states, actions, rewards $\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}_1, \mathbf{a}_1, r_1, ..., \mathbf{s}_t, \mathbf{a}_t, r_t, ..., \mathbf{s}_{T-1}, \mathbf{a}_{T-1}, r_{T-1}, \mathbf{s}_T$, where T stands for the terminal state, then policy π_1 has a sequence of states, actions, rewards $\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}_1, \mathbf{a}_1, r_1, ..., \mathbf{s}'_t, \mathbf{a}'_t, r'_t, ..., \mathbf{s}_{T-1}, \mathbf{a}_{T-1}, r_{T-1}, \mathbf{s}_T$ such that $r'_t > r_t$ [64]. Therefore, if we want the agent to explore new policies, we can inject noise into the current policy at random time steps and eventually we should find a better policy. A common method to do this is called *ϵ-greedy exploration*, where with probability ϵ the agent takes a random action during a single time step. The value of ϵ is linearly annealed over time until the agent only exploits its current policy [65]:

Algorithm 1 ϵ-greedy exploration

1: **Variables:** policy π, $\epsilon \leftarrow 1.0$, action space A, initial state $\mathbf{s}$, linear anneal rate $\beta \in \mathbb{R}$
2: **repeat**
3: sample: $p \sim Uniform(0, 1)$
4: **if** $p \leq \epsilon$ **then**
5: sample: $\mathbf{a} \sim A$ //Sample from the action-space
6: **else**
7: $\mathbf{a} \leftarrow \pi(\mathbf{s})$ //Use the policy to take an action
8: **end if**
9: take action $\mathbf{a}$ to get state $\mathbf{s}'$
10: $\mathbf{s} \leftarrow \mathbf{s}'$
11: $\epsilon \leftarrow max(0, \epsilon - \beta)$
12: **until** termination condition

On-Policy vs. Off-Policy

On-policy reinforcement learning refers to learning information (state/action values, etc.) from the policy the agent is utilising. *Off-policy* reinforcement learning refers to learning this information independent of the policy the agent is utilising.

It is common for off-policy methods to use a *replay buffer*. The replay buffer is a random-access memory (RAM) with some maximum capacity. Data is read from

the buffer uniform-randomly and data is written in a queue until the buffer is full, in which it is written uniform-randomly. The replay buffer stores a tuples of state, action, state, reward sequences $< \mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}, r >$.

Off-policy methods are usually much more *sample efficient* than on-policy methods [49]. This is because off-policy methods can reuse information - such as information stored in the replay buffer - without having to interact with th environment. On-policy methods must interact with the environment to gain information about an agent's current policy.

2.1.2 Policy Gradient Methods

Policy gradient methods (PGM) [73] is the name for a collection of algorithms that learn an optimal policy directly without bothering to model the dynamics of the environment [72]. They avoid many of the complexities of other reinforcement learning algorithms, such as valuing states and actions. We discuss two algorithms from the class of policy gradient methods: REINFORCE and Deep Deterministic Policy Gradients (DDPG).

In addition to policy gradient methods, there are many other types of reinforcement learning algorithms that are used to solve various reinforcement learning problems. Policy gradient methods were invented to deal with continuous-action control problems, which more fundamental reinforcement learning algorithms such as SARSA or Q-learning cannot handle. These other algorithms approximate some value function such as $Q(\mathbf{s}, \mathbf{a})$ that tells the agent the value of being in state $\mathbf{s}$ and taking action $\mathbf{a}$ - this can be used to move directly (value iteration) or learn a optimal policy (policy iteration). But, a problem arises when the agent wants to take the *optimal* action in a given state for *non-discrete* actions spaces or *very large* discrete action spaces. Taking the optimal action requires performing some maximisation over the action values, which requires large computation and in some cases it is even impossible.

For example, the walking environment we used has an agent controlled through a four-dimensional continuous-action space. Maximising over all the actions in this continuous space is problematic unless the action space is discretised, or we expend computation. Therefore, it makes sense to skip the intermediate step of approximating the values of the agent's situations and instead learn the policy function $\pi(\mathbf{s})$ directly, which tells the agent how to move, given the current state.

REINFORCE

The REINFORCE algorithm [83] is an unbiased on-policy algorithm which approximates a return over the actions via Monte Carlo sampling [72] of the expected return.

Popular reinforcement learning algorithms usually incorporate some value function to evaluate the states and actions that an agent can take to tell the agent how good it is to be in a state, relative to all the other states. The agent favours actions that result in high-value states, with some minor exploration to allow the agent to explore other possibilities. The traditional *Policy Gradient Method* (PGM) throws all of this away. By removing this value function we avoid having to fit a function to the

value data, but we remove the ability for the policy to evaluate itself and determine how to improve. Luckily, we have not forgotten about the rewards! In a sense, these are the source of the action-state values, but working with the immediate rewards is much faster than approximating a function over the state-action values. So how can we pair the policy and rewards to culminate in an improvement for the policy?

As with most reinforcement learning algorithms, we can pose the problem as a supervised learning algorithm over a dataset our agent generates as it explores the environment. For REINFORCE, we use a stochastic policy function which is formulated as $\pi : S \times A \rightarrow [0,1]$ where our input is some state $\mathbf{s} \in S$ and we output some probability distribution over our possible actions $\mathbf{a} \sim A$. For this problem we have a discrete number of actions (four to be exact) where each action is a vector of continuous values, $\mathbf{a} \in \mathbb{R}^N$. If we satisfy the two requirements that (1) our policy function is continuous and differentiable for all values and (2) the output of our policy function is some probability distribution over the actions, then we can use maximum (log) likelihood estimation (MLE) to calculate a gradient in which to improve our policy. The first requirement can be handled by using any differentiable function-approximator, in this case we will be using a feed-forward neural network. For the second requirement, we are working with a discrete number of continuous action values, therefore a fitting representation is to use a Gaussian distribution to approximate the distribution over each action value. Formally, for some action a_i we have $a_i \sim \mathcal{N}(\mu_{a_i}(s), \sigma^2)$, where $\mu_{a_i}(\mathbf{s}) = f(\mathbf{s})$ is some function given the input $\mathbf{s}$. The variance of the distribution σ^2 can either be constant over all actions or, similar to the mean μ_{a_i}, a function of the state: $\sigma^2(\mathbf{s}) = g(\mathbf{s})$. For our model we use $\mu_{a_i}(\mathbf{h})$ and $\sigma^2_{a_i}(\mathbf{h})$ where $\mathbf{h}$ is the final hidden layer of a neural network, and therefore the parameters of the distribution are a non-linear aggregation of the final hidden layer values.

To actually compute the gradient of our policy function, we need some metric in which to evaluate our policy and tell it, "change your probabilities so that you do that action more often" if the action was good, or, "change your probabilities so that action happens less" if the action was bad. A suitable policy metric is the *mean reward* over time because it is invariant to the initial state of the policy and works well for continuous state spaces. We define some function $J : N \rightarrow R$ that outputs a real value given a policy function parameterised by $\boldsymbol{\theta}$. We define this as: $J(\boldsymbol{\theta}) = \sum_{\mathbf{s} \in S} d(\mathbf{s}) \sum_{\mathbf{a} \in A} \pi(\mathbf{a}|\mathbf{s}) r_{\mathbf{s},\mathbf{a}}$. By finding the gradient $\nabla J(\boldsymbol{\theta}) = (\frac{d}{d\theta_0} J(\boldsymbol{\theta}), \frac{d}{d\theta_1} J(\boldsymbol{\theta}), ..., \frac{d}{d\theta_N} J(\boldsymbol{\theta}))$ we can use our favourite optimisation method and improve our policy along the gradient: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla J(\boldsymbol{\theta}_t)$. For a neural network this will incorporate back-propagation (see Section 2.2) to update the weight parameters, but the underlying update is still the same. Computing the gradient of $J(\boldsymbol{\theta})$ we get the result $\nabla J(\boldsymbol{\theta}) = \sum_{\mathbf{s} \in S} d(\mathbf{s}) \sum_{\mathbf{a} \in A} \nabla \pi(\mathbf{a}|\mathbf{s}) r_{\mathbf{s},\mathbf{a}}$. Calculating $\nabla \pi(\mathbf{a}|\mathbf{s})$ requires a subtle trick:

$$\nabla \pi(\mathbf{a}|\mathbf{s}) = \frac{\pi(\mathbf{a}|\mathbf{s})}{\pi(\mathbf{a}|\mathbf{s})} \nabla \pi(\mathbf{a}|\mathbf{s})$$

$$\nabla \pi(\mathbf{a}|\mathbf{s}) = \pi(\mathbf{a}|\mathbf{s}) \frac{\nabla \pi(\mathbf{a}|\mathbf{s})}{\pi(\mathbf{a}|\mathbf{s})}$$

$$\nabla \pi(\mathbf{a}|\mathbf{s}) = \pi(\mathbf{a}|\mathbf{s}) \nabla \log \pi(\mathbf{a}|\mathbf{s})$$

we can then substitute into $\nabla \log \pi(\mathbf{a}|\mathbf{s})$ the appropriate probability distribution

function. We used a univariate Gaussian, therefore we get:

$$\nabla J_i(\boldsymbol{\theta}) = \sum_{\mathbf{s}\in S} d(\mathbf{s}) \sum_{\mathbf{a}\in A} \pi(\mathbf{a}|\mathbf{s}) \frac{(a_i - \mu_{a_i}(s))\nabla\mu_{a_i}(s)}{\sigma^2} r_{\mathbf{s},\mathbf{a}}$$

Combining all the pieces together we get:

$$\nabla J(\theta) = \sum_{s\in S} d(s) \sum_{a\in A} \pi_\theta(a|s)\nabla \log \pi_\theta(a|s)$$

If we look at a single trajectory for return G(t):

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \log \pi(\mathbf{a}|\mathbf{s})G(t)]$$

We can generalise this algorithm by bringing back the state-action value to provide a better prediction for the return at time t:

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \log \pi_\theta(a|s)Q^{\pi_\theta}(s,a)]$$

This is also known as the *Policy Gradient Theorem*. For our initial experiments we used the *Monte Carlo* method for approximating $Q(\mathbf{s},\mathbf{a})$ by using the sample return G_t over multiple trajectories. This avoided our previous concern of having to spend resources approximating a value function. Monte Carlo also has the added benefit of having low-bias which is important for stable learning and improvement of the policy over time.

Exploration on the current policy in REINFORCE comes from sampling the distribution over the action output. As the variance $\sigma^2_{a_i}$ is reduced, the exploration becomes closer to the mean output μ_{a_i}, which is a nice property for reducing exploration and reverting to exploitation.

When using a neural network to approximate the policy, we used an architecture with two output layers, one for $\boldsymbol{\mu}_a(\mathbf{h})$ and one for $\boldsymbol{\sigma}^2_a(\mathbf{h})$. More specifically, $\boldsymbol{\mu}_a = tanh(\boldsymbol{h})$, which constrains the mean between the minimum and maximum action value $[-1,1]$, and $\sigma_a = \exp(linear(\boldsymbol{h}))$, which constrains the standard deviation to always be positive.

Actor-Critic Methods

Actor-Critic (AC) methods [72] combine the benefits of value-based methods and policy-based methods. Specifically we have an *actor* model that learns some policy for the agent, and a *critic* model which tells the 'actor' how to improve itself. In this report we are only concerned with actor models that utilise policy gradient methods to learn a policy model. The actor learns through insight from the critic, while the critic uses the environment interactions of the actor to update the quality of its evaluations. The architecture for the AC model we will be using in this report is depicted below

The AC methods can encompass a large domain of reinforcement learning algorithms. Therefore, for the sake of clarity we present an abstract AC algorithm below. This will be enhanced upon by the Deep Deterministic Policy Gradients algorithm in the next section.

Algorithm 2 The REINFORCE Algorithm

Initialise: policy network π_θ with output $\boldsymbol{\mu}$ and $\boldsymbol{\sigma^2}$
while not converged **do**
  for e in epochs **do**
    start new environment E_e
    sample trajectory $\{\mathbf{s}_0, \mathbf{a}_0, r_1, ..., \mathbf{s}_{T-1}, \mathbf{a}_{T-1}, r_T\}$
    for r_t in sampled $\{r_T,...,r_1\}$ **do**
      if $t == T$ **then**
        $G(t) = r_t$
      else
        $G(t) \leftarrow r_t + G(t+1)$
      end if
    end for
    $\nabla J(\theta) = \frac{1}{T} \sum_{t=1}^{N} \sum_{a_i \in A} \frac{(a_i - \mu_{\mathbf{a}_i}(\mathbf{s}_t))\nabla \mu_{\mathbf{a}_i}(\mathbf{s}_t)}{\sigma^2_{\mathbf{a}_i}(\mathbf{s}_t)} G(t)$
    backpropagate π_θ with $\nabla J(\theta)$
  end for
end while

Algorithm 3 General Actor Critic Algorithm

Initialise: actor π, critic Q
while not converged **do**
  for e in epochs **do**
    start new environment E_e
    sample $\{\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}_1, \mathbf{a}_1, r_1, ..., \mathbf{s}_{T-1}, \mathbf{a}_{T-1}, r_{T-1}, \mathbf{s}_T\} \sim \pi$ //Run the policy through the environment for some time steps (could be 1)
    Run critic: compare critic predictions of the returns $Q(\mathbf{s}_0, \mathbf{a}_0), Q(\mathbf{s}_1, \mathbf{a}_1), ..., Q(\mathbf{s}_T, \mathbf{a}_T)$ to the actual returns $G_0, G_1, ..., G_T$ according to some error metric J
    Update critic: update parameters of Q using the error metric J
    Update actor: update the parameters of π using the new critic Q
  end for
end while

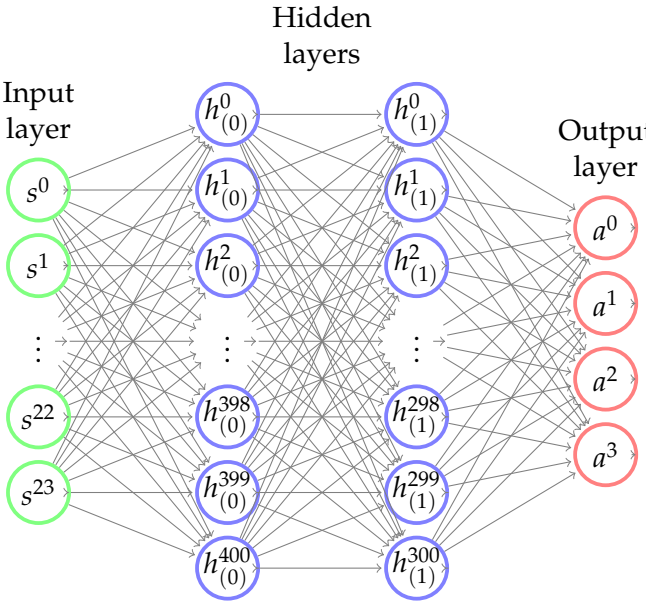


FIGURE 2.2: Actor network for DDPG algorithm (see below).

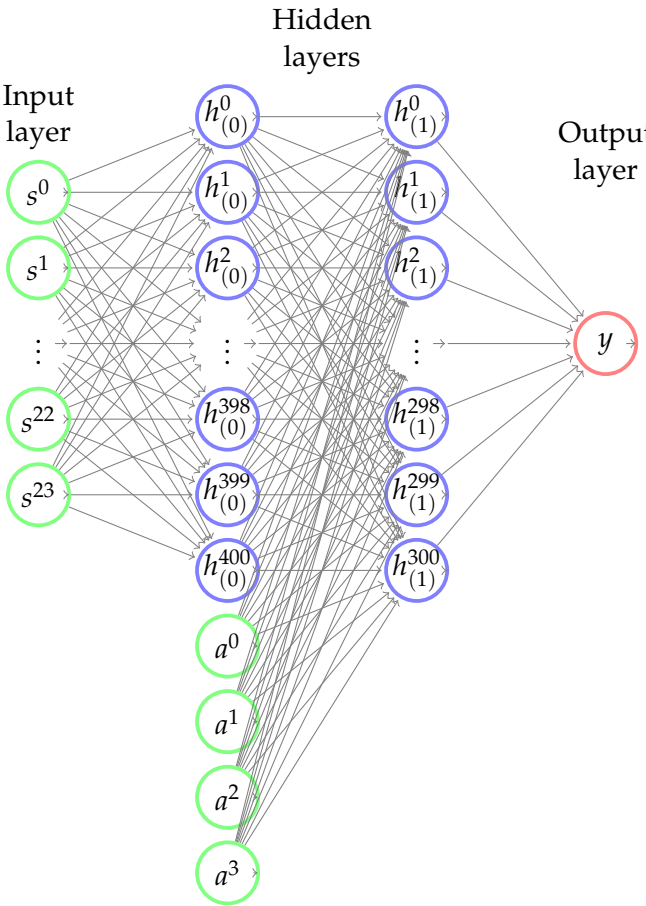


FIGURE 2.3: Critic network for DDPG algorithm (see below). The action vector is introduced during the second hidden layer.

Deep Deterministic Policy Gradients

The *Deep Deterministic Policy Gradients* (DDPG) algorithm [40] is a special case policy gradient method that allows for deterministic continuous-action control. It

also uses an Actor-Critic model to reduce variability. REINFORCE is a fitting example of where well-formed theory does not always entail good results. In practise, REINFORCE suffers from high variance when used for problems with long time spans and complicated state spaces [76]. Later in the report we will show that we struggled with getting consistently good results from REINFORCE.

What is interesting is if we look at the limiting case of the *Policy Gradient Theorem* described earlier. In this theorem, the policy was a probability distribution over actions given the state. Specifically a Gaussian distribution was used to model the action space. As $\sigma \rightarrow 0$ for a Gaussian distribution, intuitively an action sampled from this Gaussian will be within ϵ of the mean μ, where ϵ will approach zero. At zero we can say that our stochastic policy is now a deterministic policy [66]. From this limiting case, we get the *Deterministic Policy Gradient Theorem*, which is derived from the *Policy Gradient Theorem* as a limiting case:

Definition 2.1.2 (Deterministic Policy Gradient Theorem) *Given that:*

$$lim_{\sigma \rightarrow 0} \nabla J(\pi_\theta) = \nabla J(\mu_\theta)$$

Then:

$$\nabla J(\theta) = \nabla \mu_\theta Q^{\pi_\theta}(s, a)$$

This is an incredible result because it means we can map a deterministic policy function $\pi : S \rightarrow A$ and take the gradient directly, scaled by the size of the expected return $Q(\mathbf{s}, \mathbf{a})$. This works nicely with backpropagation through a neural network, where we can output $\boldsymbol{\mu_\theta} \rightarrow \mathbb{R}^N$, whose gradient is simply a vector of ones and therefore our error to backpropagate is only $Q(\mathbf{s}, \mathbf{a})$.

We run into trouble when dealing with convergence and exploration [66] but there are four additions to the algorithm which can mitigate these issues. These were introduced by T. Lillicrap et al [40] to help apply deterministic policy gradients to deep neural networks, hence the name *Deep Deterministic Policy Gradients*. Without too rigorous an analysis, we will outline the four additions and the problems they solve:

1. PROBLEM: approximating $Q(\mathbf{s}, \mathbf{a})$ via Monte Carlo sampling is inefficient and not likely to be the true return because the policy changes online.
   SOLUTION: add a *critic* network $Q^{\mathbf{w}}(\mathbf{s}, \mathbf{a})$ which learns the true value of $Q(\mathbf{s}, \mathbf{a})$ by minimising the error of the *Bellman Equation* (see Section 2.1) $[r_t + \gamma Q^{\mathbf{w}}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q^{\mathbf{w}}(\mathbf{s}_t, \mathbf{a}_t)]^2$.

2. PROBLEM: training $Q^{\mathbf{w}}(\mathbf{s}, \mathbf{a})$ online results in sequential episodes being correlated. We need our training data to be identically and independently distributed (i.i.d.) for training to converge.
   SOLUTION: implement a *replay buffer* which stores the last M episodes $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ and sample a batch from this buffer at each time step. We still get the speed of training online, but now our data is i.i.d..

3. PROBLEM: the error function for $Q^{\mathbf{w}}(\mathbf{s}, \mathbf{a})$ bootstraps itself: $[r_t + \gamma Q^{\mathbf{w}}(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) - Q^{\mathbf{w}}(\mathbf{s}_t, \mathbf{a}_t)]^2$, which will incorporate bias into the training and cause divergence and instability [65].
   SOLUTION: introduce a *target network* $Q^{\tau}(\mathbf{s}, \mathbf{a})$ that gradually incorporates the weights of $Q^{\mathbf{w}}(\mathbf{s}, \mathbf{a})$ such that $W_{Q^{\tau}(\mathbf{s},\mathbf{a})} = (1 - \tau) W_{Q^{\tau}(\mathbf{s},\mathbf{a})} + \tau W_{Q^{\mathbf{w}}(\mathbf{s},\mathbf{a})}$, where

$\tau << 1$ (recommended value is $\tau = 0.001$). Like two ice climbers roped together as they ascend a mountain, one tugs the other on the correct path and tests for safe footing, the other follows behind from a stable route.

4. PROBLEM: because DPPG does not have a stochastic policy, we cannot incorporate exploration via the probability distribution as we did with REINFORCE.
   SOLUTION: DDPG is off-policy so we can inject noise directly into the environment as the agent actuates $\pi'(s) = \pi(s) + \mathbb{N}$ where $\mathbb{N}$ is some sampled noise. During test time we remove this noise.

Additionally, because the policy network (also referred to as the *actor* network) will be trained with the true state-action value estimate, $Q^w(s, a)$, we avoid the same problems with convergence by introducing a target policy network $\pi^\tau(s)$ as well such that: $W_{\pi^\tau(s)} = (1 - \tau)W_{\pi^\tau(s)} + \tau W_{\pi(s)}$. τ is the same as for the *critic* network.

Algorithm 4 The Deep Deterministic Policy Gradient Algorithm

Initialise: policy 'actor' network $\pi^\tau_{\theta_1(\mathbf{s})}$, q-value 'critic' network $Q_{\theta_2}(\mathbf{s}, \mathbf{a}_,)$ target networks $\pi^\tau_{\theta_1^\tau}(\mathbf{s})$ and $Q^\tau_{\theta_2^\tau}(\mathbf{s}, \mathbf{a}_,)$ buffer b, $N(t)$ noise function
while not converged **do**
  for e in epochs **do**
    start new environment E_e
    $\mathbf{s}_t \leftarrow$ init. state
    while E_e not terminated **do**
      $\mathbf{a}_t \leftarrow \pi^\tau_{\theta_1(\mathbf{s}_t)}$
      if train **then**
        $\mathbf{a}_t \leftarrow \mathbf{a}_t + N(t)$
      end if
      $\mathbf{s}_{t+1}, r_{t+1} \leftarrow$ TakeAction($\mathbf{a}_t$)
      b.add(<$\mathbf{s}_t$, $\mathbf{a}_t$, r_{t+1}, $\mathbf{s}_{t+1}$>)
      sample batch $D \leftarrow < S_0, A, R, S_1 >$ from b
      $\nabla J_2(\theta_2) \leftarrow \nabla[\frac{1}{T}\sum_{t=1}^{T}(r_t + \gamma Q^\tau(\mathbf{s}_{t+1}, \pi^\tau_{\theta_1^\tau}(\mathbf{s}_{t+1})) - Q_{\theta_2}(\mathbf{s}_t, \mathbf{a}_)^2]$
      $\nabla J_1(\theta_1) \leftarrow \frac{1}{T}\sum_{t=1}^{T} Q(\mathbf{s}_t, \pi^\tau_{\theta_1(\mathbf{s}_t)})\nabla log(\pi(\mathbf{s}_(t))$
      $\pi^\tau_{\theta_1(\mathbf{s})} \leftarrow \pi^\tau_{\theta_1(\mathbf{s})} + \alpha\nabla J_1(\theta_1)$
      $Q_{\theta_2}(\mathbf{s}, \mathbf{a}_\leftarrow)Q_{\theta_2}(\mathbf{s}, \mathbf{a}_+)\alpha\nabla J_2(\theta_2)$
      $Q^\tau_{\theta_2^\tau}(\mathbf{s}, \mathbf{a}) \leftarrow (1 - \tau)Q^\tau_{\theta_2^\tau}(\mathbf{s}, \mathbf{a}) + \tau Q_{\theta_2}(\mathbf{s}, \mathbf{a})$
      $\pi^\tau_{\theta_1^\tau}(\mathbf{s}) \leftarrow (1 - \tau)\pi^\tau_{\theta_1^\tau}(\mathbf{s}) + \tau\pi^\tau_{\theta_1(\mathbf{s})}$
    end while
  end for
end while

2.1.3 Black-Box Algorithms

Black-box algorithms have been commonly used to successfully solve a range of problems. These algorithms usually suffer from high-computation costs, but have the benefit of being able to solve most general optimisation problems, even if the function being optimised is non-differentiable. We are interested in exploring the subset of black-box algorithms called *Evolutionary Algorithms* which take inspiration

from biological evolution to optimise some objective function. Specifically, we focus on the *Natural Evolutionary Strategies* algorithm [82] [60] - referred to as just the *Evolutionary Strategies* (ES) throughout this report.

There are a few common evolutionary algorithms used to learn neural network architectures, some of which include *Covariance Matrix Adaptation* (CMA-ES), *Neuro-Evolution of Augmenting Topologies* (NEAT), and various *Genetic Algorithms* (GAs). Where ES triumphs over them all is in computational and exploratory power [25]. For example, CMA-ES suffers from large computation time for neural networks with large hyperparmaters [25] and GAs are not designed for large continuous-parameter spaces.

In addition, we also implement a baseline algorithm to show how both ES and PGM perform relative to a simple implementation. This algorithm, called the *Cross Entropy Method*, is also a black-box algorithm that optimises some given objective function.

Evolutionary Strategies

Evolutionary strategies is a gradient descent approximation method with a loose biological analogy, developed by Rechenberg in the 1970s [55]. The basis of evolution require three conditions: replication, variation, and competition [17].

During gradient descent (or in our case, gradient *ascent* because we want to maximise the amount of return), we usually favour directions of steepest descent, as these directions are most likely to lead to a minimum solution the quickest. Imagine we are on the peak of a hill and we want to get to the bottom following the fastest path possible. Each step we take down this hill is of equal magnitude - let's say the extent to which your legs stretch. Naturally, the fastest path will be the steepest path, as it provides the most direct route to the bottom (envision the extreme case where it is just a cliff, one step off the edge and you are at the bottom). We can invent a device on a pole that we hold out horizontally from us and that will measure how far away the ground the device is from a position We perform this measuring N number of times in N-different directions and then we choose the direction which is the steepest based on the device's reading. In this scenario there are a couple of factors that must be accounted for: How many readings do you take? Too many would mean hours before you can take your first step; too few and you may miss detecting the steepest route. How far out from your position do you take device readings from? If the readings are too close, there will be little difference between readings in different directions; too far and a direction you believe looks promising actually contains an upward jut before the reading. Finally, a more robust approach than merely choosing the best reading of the device before each step, is to take a weighted average of all the directions we have sampled, weighted by the gradient of the step. In this case, directions with small gradients will contribute least to our next step, whereas directions with big gradients will contribute the most.

Once this analogy is understood, translating it to the realm of evolutionary strategies requires transforming everything into its mathematical equivalent. We represent our position on a D-dimensional hill as a vector $\boldsymbol{\theta} \in \mathbb{R}^D$. Our 'readings' are represented by a multivariate Gaussian distribution $\mathcal{N}(0, \mathbb{I}_D) \times \sigma$ with variance σ^2, centered around our position. Each reading is a sample from this Gaussian, $\boldsymbol{\epsilon}_i \sim \mathcal{N}(0, \mathbb{I}_D)\sigma$, resulting in many close readings and some distant readings. A

nice balance. The 'device' reading is some objective function $F : M_{\boldsymbol{\theta}} \rightarrow \mathbb{R}$ mapping some model, parameterised by our sampled position $\boldsymbol{\theta}$, to a reward value. The objective function is similar to the concept of return in the reinforcement learning problem in that it tells how much total expected return we will get by moving to the new position. The magnitude of the reward is analogous to the height of the hill. So for a single reading stretching out from our position, we get the reward value: $r_i = F(\boldsymbol{\theta} + \boldsymbol{\epsilon}_i)$, where $F(\mathbf{x})$ is the objective function. We can then take an weighted average over N samples from our position to compute the approximate next best position to move to:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \frac{1}{N}\sum_{i=1}^{N} r_i \boldsymbol{\epsilon}_i \\ &= \boldsymbol{\theta}_t + \mathbb{E}_{\boldsymbol{\epsilon}\sim\mathcal{N}(0,\mathbb{I}_D)\sigma}[r\boldsymbol{\epsilon}]\end{aligned}$$

Instead of jumping directly to the new position, we can treat this as an approximation of the gradient of the surface we are optimising. We therefore can take finitely small steps in the approximated direction. Currently, the amount we move is both proportional to the size of the rewards we receive - note how Equation **??** scales to extreme values as r_i increases - and the size of the Gaussian distribution we placed around our position, σ^2. Both of these factors can be removed by normalising each one out such that:

$$\begin{aligned}\nabla J\boldsymbol{\theta} &= \frac{1}{N}\sum_{i=1}^{N} r_i \frac{\boldsymbol{\epsilon}_i}{\sigma} \\ &= \frac{1}{N\sigma}\sum_{i=1}^{N} r_i\end{aligned}$$

And we can perform *stochastic gradient descent* (SGD) like so:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha\nabla J\boldsymbol{\theta} \\ \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \alpha\frac{1}{N\sigma}\sum_{i=1}^{N} r_i\end{aligned}$$

Where $\alpha << 1$ is the step-size. It is possible to also use other optimisation methods, such as Adam [35], in place of SGD.

OpenAI alludes to evolutionary strategies having two interesting benefits [60]: the analytical gradient is not calculated which protects against the exploding gradients problem that are common in recurrent architectures and that the variance of ES is independent of the time steps T of a trajectory, minimising the variance. Both properties are useful and provide interesting avenues of research. Although we 'blindly' approximate the analytical gradient which can be less useful than some heuristic to guide our gradient, it is less computationally expensive than backpropagation which is required for the analytical gradient to be computed. OpenAI asserts that the removal of backpropagation is up to three times faster [60]. Additionally, because the architecture does not need to be differentiable, some new model architectures can be explored, for example Jordan Nets [33], binary neural networks [13], or attention mechanisms [86]. We will explore some of these concepts later.

Algorithm 5 The Evolutionary Strategies Algorithm

Variables: params $\boldsymbol{\theta}$, noise σ, population size n, learning-rate α, $F(\cdot)$ function that converts the parameters to a neural network and runs the agent through the environment
for e in epochs **do**
  for i in $\{1, ..., n\}$ **do**
    $\epsilon_i \sim \mathbb{N}(1, \mathbb{I})\sigma$
  end for
  for i in $\{1, ..., n\}$ **do**
    $r_i \leftarrow F(\boldsymbol{\theta} + \epsilon_i)$
  end for
  $normalise(\{r_1, ..., r_n\}$
  $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \frac{1}{n\sigma} \sum_{i=1}^{n} r_i \cdot \epsilon_i$

end for
procedure F($\boldsymbol{x}$)
  $N_{\boldsymbol{x}}() \leftarrow$ convert $\boldsymbol{x}$ to neural network
  $R \leftarrow 0$
  start new environment E
  $\mathbf{s}_t \leftarrow$ init. state
  while E not terminated **do**
    $\mathbf{a}_t \leftarrow N_{\boldsymbol{x}}(\mathbf{s}_t)$
    $\mathbf{s}_{t+1}, r_{t+1} \leftarrow$ TakeAction($\mathbf{a}_t$)
    $R \leftarrow R + r_{t+1}$
    $\mathbf{s}_t \leftarrow \mathbf{s}_{t+1}$
  end while
  return R
end procedure

Cross Entropy Method (The Baseline)

For our baseline we employ the Cross Entropy Method (CEM) [43], which is a simple and efficient approach for combinatorial optimisation as well as reinforcement learning [18] tasks, and advertises a intuitive implementation with good results in simple domains. There are subtle similarities between CEM and ES, but the two methods are different. Whereas ES utilises gradient descent and approximates the gradient via a population of parallel agents, CEM samples independent distributions over the individual parameters and performs greedy 'hill climbing' to improve the parameters. We now outline the CEM algorithm in further detail.

We have some function $f : \mathbb{R}^n \rightarrow \mathbb{R}^k$, $f(\mathbf{s}; \boldsymbol{\theta}) = \mathbf{a}$ that takes a vector, $\mathbf{s}$, representing the state of the environment and transforms it via the constant parameter vector, $\boldsymbol{\theta}$, into the action vector, $\mathbf{a}$. For our experiments we use a linear transformation such that $f(\mathbf{s}) = \boldsymbol{\theta}^T \mathbf{s}$. We treat our parameters as Gaussian-distributed with mean $\boldsymbol{\theta}$ and variance, $\mathbf{v}$. $\boldsymbol{\theta}$ and $\mathbf{v}$ are initialised to some constant value for all elements in the vectors. At each step in the algorithm, we draw N samples from the parameter distribution, $X = \mathbf{x_n} \sim \mathbb{N}(\boldsymbol{\theta}, \mathbf{v})_{n=1}^{N}$, $n = 0, 1, .., N$ and compute $R(f(\mathbf{s}; x_n))$ where $R(.)$ is an objective function. For this problem, $R(.)$ indicates running one episode in the environment, and computing the discounted return, G. The samples with

the top-m highest discounted returns are kept and the parameters, $\boldsymbol{\theta}$, are recalculated via the sample mean: $\boldsymbol{\theta} = \frac{1}{m}\sum_{i=1}^{m} \mathbf{x_i}$. The variance, $\mathbf{v}$, is not sampled because the high-variability in the sampled parameter space will cause the variances to explode/vanish. Instead the variances are either kept constant, or annealed at each time step so that $\mathbf{v}_{t+1} \leftarrow \gamma \mathbf{v}_t$, $0 \leq \gamma \leq 1$. For the full algorithm see Algorithm 6.

Algorithm 6 The Cross Entropy Method

Variables: state $\mathbf{s}$, params $\boldsymbol{\theta}$, vars $\mathbf{v}$, $\epsilon = 0$, $e = 0$
repeat
  for i in $\{1, ..., n\}$ **do**
    $\mathbf{x_i} \sim \mathbb{N}(\boldsymbol{\theta}, \mathbf{v})$
  end for
  for i in $\{1, ..., n\}$ **do**
    $r_i \leftarrow R(f(\mathbf{s}, x_i))$
  end for
  sort increasing: $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_N}$ using $r_1, r_2, ..., r_N$
  take top-m samples and compute params: $\boldsymbol{\theta}_{t+1} \leftarrow \frac{1}{m}\sum_{i=1}^{m} \mathbf{x_i}$
  $\mathbf{v}_{t+1} \leftarrow \gamma \mathbf{v}_t$
  $\epsilon \leftarrow |\frac{1}{N}\sum_{i=1}^{N} \mathbf{r_i} - \epsilon|$
  $e \leftarrow e + 1$
until e $> N$ or $\epsilon < c$

The cross entropy method has been successful for a variety of reinforcement learning problems [44], a handful of interesting ones include learning to play Tetris [29] and other arcade games [61], and assisting the control of a unmanned aerial vehicle (UAV) [47]. CE has been less popular recently because it has a predisposition towards local optima. For simpler problems this is usually acceptable. CE is a good baseline because it is a bare-minimum optimisation strategy - by performing better than it, the algorithms show they are more effective than adroit hill-climbing

2.2 Neural Networks

The use of neural networks in reinforcement learning problems has exploded over the past decade and has produced a new branch of reinforcement learning known as *deep reinforcement learning* (deep-RL). This is not to say that the approach of using neural networks with reinforcement learning algorithms is a recent conception. In the 90s, Tesauro demonstrated a Backgammon-playing agent that became as good as the world's best Backgammon players [74]. But since this achievement, the field of deep-RL has made many advances.

Here we provide some background on the neural network model, why it is useful, and the specifics of how neural networks are used in reinforcement learning.

2.2.1 The Multi-Layer Perceptron

What is colloquially referred to as a 'neural network' is actually a multi-layer perceptron (MLP); a nod of recognition to its turbulent past. The MLP was rigorously studied by the Parallel Distributed Processing group and its modern implementation is largely because of their work [58]. The most common MLP is the feed-forward neural network (FFNN).

The structure of an FFNN is a series of densely-connected nodes in an directed acyclic, weighted graph. Nodes a divided into D 'hidden layers', such that the FFNN is composed of a set of D layers with a set of $D+1$ weighted outputs from each layer, including input weights: $FNN :< \{\boldsymbol{L}_1, \boldsymbol{L}_2, ..., \boldsymbol{L}_D\}, \{\boldsymbol{W}_1, \boldsymbol{W}_{1,2}, \boldsymbol{W}_{2,3}, ..., \boldsymbol{W}_{D-1,D}, \boldsymbol{W}_D\} >$. The weights, $\boldsymbol{W}_{j,i}$, between two 'hidden' layers, $\boldsymbol{L}_i, \boldsymbol{L}_j$, are represented by a matrix $\mathbb{R}^{|\boldsymbol{L}_i|} \times \mathbb{R}^{|\boldsymbol{L}_j|}$ and the output from $\boldsymbol{L}_i$ is fed forward to $\boldsymbol{L}_j$ through a matrix multiplication, the addition of a bias vector $\boldsymbol{b}_i \in \mathbb{R}^{|\boldsymbol{L}_i|}$, and a non-linear 'activation' function $f(.)$:

$$\boldsymbol{L}_i^{in} = f(\boldsymbol{W}_{j,i}^T \boldsymbol{L}_j^{out} + \boldsymbol{b}_i)$$

The input to the FFNN is a vector $\boldsymbol{x} \in \mathbb{R}^N$ which is fed into the first hidden layer $\boldsymbol{L}_1$ via a similar process:

$$\boldsymbol{L}_1^{in} = f(\boldsymbol{W}_1^T \boldsymbol{x} + \boldsymbol{b})$$

Where the bias of the input is $\boldsymbol{b} \in \mathbb{R}^{|\boldsymbol{L}_1|}$ and the input weights are a matrix $\boldsymbol{W}_1 \in \mathbb{R}^N \times \mathbb{R}^{|\boldsymbol{L}_1|}$. The output $\boldsymbol{y} \in \mathbb{R}^M$ of a FFNN is the output of the final layer $|L_D|$:

$$\boldsymbol{y} = f(\mathbf{W}_D^T \boldsymbol{L}_D^{out} + \boldsymbol{b}_D)$$

Where the bias of the output is $\boldsymbol{b} \in \mathbb{R}^M$ and the output weights are a matrix $\boldsymbol{W}_D \in \mathbb{R}^M \times \mathbb{R}^{\boldsymbol{L}_D}$. Trivially, this can be extended to a 'batch' of inputs. This is explained well in *Deep Learning* [20] by I. Goodfellow *et al*.

Backpropagation Algorithm

Backpropagation is the algorithm used to adjust the weights of the MLP so that for a given input it produces the desired output [57].

The MLP architecture feeds-forward from layer-to-layer, where the final layer outputs the result. If the MLP had linear-activation functions, or no activation functions at all, it would be equivalent to performing linear regression. If the MLP had no hidden layers, that is no additional layers other than the input and output, it would be the same as logistic regression. Therefore, because the MLP has at least one hidden layer, a new training algorithm is required. Backpropagation is the algorithm used to adjust the weights of the MLP so that for a given input it produces the desired output. We define an objective function $E : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$ such that for a *given* input vector $\mathbf{x} \in \mathbb{R}^D$, an *expected* output vector $\mathbf{y} \in \mathbb{R}^N$, and a *predicted* output vector from the MLP when it is given the input $\hat{\mathbf{y}} = f(\mathbf{x})$, where $f(.)$ represents the MLP, we can compute an *error value* between the *predicted* output and *expected* output: $E(\mathbf{y}, \hat{\mathbf{y}})$. A large error value means the MLP did poorly in predicting the output, a small error value means the MLP did well in predicting the output. Using partial differentiation, the *backpropagation algorithm* traces the influence of each weight in the MLP to the *error value* and provides the gradient direction to update each weight in order to reduce the error for the *expected* output. We define the 'error signals' that are sent from a down-stream layer to the layer before it as:

$$\delta_{i+1} = \frac{\delta E}{\delta \hat{\boldsymbol{y}}}, \text{ where } \hat{\boldsymbol{y}} \text{ is the output of layer } \boldsymbol{L}_i$$

We can then compute the gradients for layer $\boldsymbol{L}_i$ with respect to this error signal from layer $\boldsymbol{L}_{i+1}$:

$$\begin{aligned}\frac{\delta E}{\delta \boldsymbol{b}} &= \delta_{i+1}\sigma'(\boldsymbol{z}_i^T \boldsymbol{W}_{i,i+1}) \\ \frac{\delta E}{\delta \boldsymbol{W}_{i,i+1}} &= \delta_{i+1}\sigma'(\boldsymbol{z}_i^T \boldsymbol{W}_{i,i+1})\mathbf{z}_i \\ \frac{\delta E}{\delta \boldsymbol{W}_{i,i+1}} &= \boldsymbol{W}_{i,i+1}^T \delta_{i+1}\sigma'(\boldsymbol{z}_i^T \boldsymbol{W}_{i,i+1}) \\ &= \delta_i\end{aligned}$$

Intuitively, the error gradients backpropagate through the network to the lowest layer. In this situation we treat the backpropagation algorithm as having a single input and output data point when computing the gradients. During training, it is common to use a batch of size B inputs and outputs:

$$X_{batch} = \begin{matrix} x_0^{(0)} & x_1^{(0)} & \dots & x_D^{(0)} \\ x_0^{(1)} & x_1^{(1)} & \dots & x_D^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ x_0^{(B)} & x_1^{(B)} & \dots & x_D^{(B)} \end{matrix} \quad y_{batch} = \begin{matrix} y_0^{(0)} & y_1^{(0)} & \dots & y_N^{(0)} \\ y_0^{(1)} & y_1^{(1)} & \dots & y_N^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ y_0^{(B)} & y_1^{(B)} & \dots & y_N^{(B)} \end{matrix} \tag{2.1}$$

Extending the backpropagation algorithm to use batches is a straightforward process, but we do not discuss it here. Again, this is well outlined in [20].

2.2.2 Neural Networks in Reinforcement Learning

In this section we will briefly touch on the common architectures of MLPs that are required to understand most deep-RL algorithms in addition to the algorithms in this report. The three natural choices of functions that an MLP can model in reinforcement learning are:

1. State Value Function, $V(\mathbf{s})$
2. State-Action Value Function, $Q(\mathbf{s}, \mathbf{a})$
3. Policy Function, $\pi(\mathbf{s})$

The state value function and state-action value function in the *Bellman Equations*, and the policy function can all be modeled using MLPs. The State Value Network (V-Network) approximates the function $V : S \to R$ by using supervised learning where the input is some state $\mathbf{s}_t$, and the output is the return G_t the agent expects to receive from the state. When training the network it is common to use either an off-line method to compute the return at the end of each episode for all states, such as in Monte Carlo sampling, or to bootstrap the State Value Network knowing that $V(\mathbf{s}_t) = r_t + \gamma V(\mathbf{s}_{t+1})$. The latter allows for on-policy training but can incorporate bias into the model [65]. The error function for a single sample is therefore $E(\mathbf{w}) = r_t + \gamma V'(\mathbf{s}_{t+1}; \mathbf{w}) - V'(\mathbf{s}_t; \mathbf{w})$ where $V'(\cdot; \mathbf{w})$ is an MLP with weights $\mathbf{w}$.

The Action-State Value Network (Q-Network) approximates the function $Q : S \times A \to R$ by using supervised learning where the input is some state $\mathbf{s}_t$ and some action $\mathbf{a}_t$ and the output is the return G_t the agent anticipates it will receive when in

the state and performing the action. As with the State Value Network, this can be trained either off-line by sampling complete episodes or on-line by bootstrapping the equation $Q(\mathbf{s}_t, \mathbf{a}_t) = r_t + \gamma Q(\mathbf{s}_{t+1}, \mathbf{a}_{t+1})$. Again, the error function for a single sample is therefore $E(\mathbf{w}) = r_t + \gamma Q'(\mathbf{s}_{t+1}, \mathbf{a}_{t+1}; \mathbf{w}) - Q'(\mathbf{s}_t, \mathbf{a}_t; \mathbf{w})$ where $Q'(\cdot; \mathbf{w})$ is an MLP with weights $\mathbf{w}$.

Lastly, we can also use an MLP to fit the policy function $\pi : S \rightarrow A$ for the deterministic case and $\pi : S \times A \rightarrow [0, 1]$ for the stochastic case. As fitting the policy function cannot be easily done with a supervised approach and is specific to a branch of RL algorithms called policy gradient methods, we outlined this in the context of the algorithms in section 2.1.2. Fitting an MLP to the states and actions of an agent is not straightforward because it is not clear how to represent a negative or positive signal for a given action.

2.3 Bipedal Walking

The ability to perform a controlled walk, whether it is with two legs or more, is a challenging control problem for engineers. Yet, paradoxically, *bipedalism* - the act of walking with two legs - is a natural phenomenon in nature. Bipedal walkers have evolved independently numerous times in the history of evolution [2], suggesting that this behaviour is optimal, or at least useful [1].

In this section we familiarise the reader with the physics and biology of a bipedal gait. The physics of a bipedal gait was used to develop accurate metrics for determining the quality of a bipedal walk (as discussed in 3.2. We found inspiration in biological control systems in the human body to design novel models which are discussed in detail in Section 2.3.2.

2.3.1 Physics of Walking

Taking a moment to introspectively analyse one's walk, and you will find some simple heuristics that describe it. In Marc Raibert's book, *Legged Robots That Balance* [54] he describes the engineering behind a human bipedal gait and we outline the key points here.

When you walk you do not sway or even fall over - instead you maintain a constant speed and a consistent posture. For this to be the case, the net forces on your body must be zero over some finite time period, such that you do not experience any acceleration. As you lift one leg forward, you are applying torque unevenly from your waist, on one side of the body. To counter this you must apply torque in the opposite direction, and hence your second leg responds by applying a backward torque of equivalent strength. This is described by the following:

$$\theta_b(t) = -\theta_f(-t) \tag{2.2}$$

$$\tau_b(t) = -\tau_f(-t) \tag{2.3}$$

Where b denotes the back leg, f denotes the front leg, $\theta(t)$ is the angle of the waist as a function of time, and $\tau(t)$ is the torque on the waist as a function of time. At

[1] An interesting example of the evolution of bipedalism is found in lizards. They use a bipedal gait when escaping predators because it allows them to move faster [69]

the halfway point between a single step, $t = 0$. Interestingly, these equalities hold for steps in unison, for example in a hopping agent. With Equation 2.2, as well as the velocity and position of the agent remaining symmetric over a time period, the walk will be in a *steady state*. This is discussed in further detail in Section 3.2. [2] More information can be found in, *Legged Robots That Balance* [54].

2.3.2 Biology of Walking

We do not know *how* and *why* humans developed the ability to walk. This is proven by the diversity of biological theories that all try to piece together the puzzle of how evolution solved this complicated control task [2] [9] [59] [56]. When developing new model architectures to create more human-like gaits, we turned to research in biology for ideas. Specifically, we were interested in how *central pattern generators* (CPGs) in the central nervous system and *muscle coactivation* on joints contribute to the human gait.

Central Pattern Generators

Studies in the field of walking dynamics in the nervous system have shown that some mammals (including humans) continue to perform walking even when the spinal cord is disconnected from the brain [34]. A system of neurons in the central nervous system called *central pattern generators* (CPGs) maintain cyclic firing of neurons that keep the rhythm of a walk (it is believed these CPGs also cause other rhythms in the body such as breathing [31] and the swinging of arms). The brain's part in walking is still significant, but the difficulties that come with orientating all the muscles in the entire body make it unlikely the brain is performing this task directly; instead the brain provides information of target positions of the respective body parts to move [11]. CPG neural networks have been designed before to try to improve gaits [19] [87] [28].

Muscle Coactivation

The major muscles in the legs - and in most of the body - are restricted to only pulling motions. Therefore, on a joint such as the knee, two sets of muscles are required: one set for pulling the knee closed, and another set for pulling the knee open. When either of these muscles are activated to control a joint, this is referred to as *agonist* muscular behaviour [42]. If, while one muscle is activated agonistically, an opposing muscle activates, this newly activated muscle is referred to as the *antagonist* [42]. Together, the two muscle make an *agonist-antagonist pair*. This is the process of muscle *coactivation* and was first identified through Lombard's paradox [41].

Muscle coactivation is believed to have a pivotal role in postural control on uneven ground [36]. The opposing contraction of muscles stabilises the joint and protects it from heavy loads. Additionally, coactivation allows for joints to be finitely manipulated by small adjustments to either the agonist or antagonist muscle [1] [7].

[2] The reader may be interested to know that the symmetrical relationship of two legs translates directly to that of having only one leg or even four legs.

Incorporating muscles into robots has been difficult [75] but has shown to have powerful properties for creating robots that walk similar to humans [52]. Therefore it is important to study how the use of muscles is advantageous over common 'robotic' servos and motors.

Chapter 3

The Walking Problem

In this section we introduce the simplified two-dimensional walking environment, as well as the *Bipedal Balancer* environment initially used to test the algorithms. At the end of this section we discuss the metrics we used to quantitatively discriminate a 'natural' bipedal gait from an 'unnatural' bipedal gait. A reflection on the effectiveness of these metrics is found in the final discussion in Chapter 8.

3.1 The Environments

The focus of this work was on a single environment modeling a simple two-dimensional bipedal walker. We also used a modified version of the environment for preliminary tests where the reward function was altered.

3.1.1 Bipedal Walker v2

For all experiments, we worked with the *Bipedal Walker v2* environment from *OpenAI gym* [48]. The *Bipedal Walker v2* environment consists of a two-dimensional agent in a horizontal world. The agent experiences conditions similar to that on Earth, including gravity and friction. The anatomy of the agent consists of a 5-sided polygon, called the *hull*, attached to two identical legs. One leg is the *back* leg, which is a lighter colour, and the other the *front* leg, which is a darker colour. Both legs are at different depth planes such that they cannot collide. A single leg is composed of two congruent rectangles; a rectangle for the thigh is connected to the hull at a joint location called the *waist*, and a rectangle for the shin is connected to the *distal* location of the thigh at a joint location called the *knee*. This is depicted in Figure 3.1.

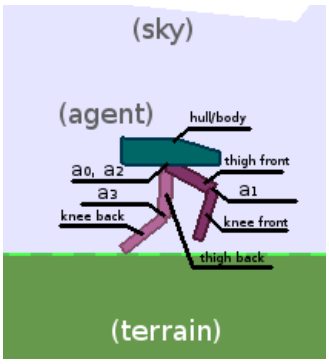


FIGURE 3.1: Labeled anatomy of the parts of the agent.

The agent walks on a terrain where its height is determined at a particular time step t according to the equation:

$$v_g(t) \leftarrow \alpha_0 * v(t-1) + 0.01 * sign(H - y_g(t)) + \alpha_1 U(-1,1) * a_g \tag{3.1}$$

$$y_g(t) \leftarrow y_g(t-1) + v_g(t) \tag{3.2}$$

where $v_g(t) \in \mathbb{R}$ is the velocity of the terrain height at time t, $H \in \mathbb{R}$ is the starting height of the terrain, $y_g(t) \in \mathbb{R}$ is the height of the terrain at time t, $a_g \geq 0$ is the scaling factor of acceleration of the terrain position, $U(-1, 1)$ is a uniform sample in $(-1, 1)$, and $\alpha_0 = 0.8, \alpha_1 = 30.0$. The terrain is constrained such that the first $T = 20$ time steps of generation are constrained to height H - this makes it easier for the agent to initially start learning.

The environment state is structured so that the observation data the agent receives is a 24-feature vector consisting of proprioceptive sensors as well as information about the velocity and acceleration of the agent.

$$\boldsymbol{s} = (s_h, \omega_h, s_x, s_y, \theta^b_w, s^b_w, \theta^b_k, s^b_k, contact(leg_b), \theta^f_w, s^f_w, \theta^f_k, s^f_k, contact(leg_f), \mathbb{L}_{10})$$

Where s_x, s_y represent the x and y speed, ω represents angular momentum, θ represents angle, $contact(leg_i)$ is a contact sensor for leg i which is 1 when the leg is in contact with the ground and 0 otherwise, and L_{10} represents a vector of 10 different lidar sensor inputs. f denotes the front leg, b denotes the back leg, w denotes the waist, k denotes the knee.

The agent acts with a four-feature action vector of continuous values that represent the torque of its four leg joints; two joints, waist and knee, for each leg. Continuous-action locomotion tasks are considered some of the most challenging in reinforcement learning benchmarks [18]. The action vector, $\mathbf{a} = (a_0, a_1, a_2, a_3)$, represents the torque force τ applied to each joint such that it is mapped to the anatomy as: $(\tau^b_w, \tau^b_k, \tau^f_w, \tau^f_k)$ (as shown in Figure 3.1). At each time step when an agent performs an action, the gym API returns the corresponding reward the agent receives from the environment as well as the next state the environment is in. A small forward movement (from left to right) results in a small reward; applying torque to the joints results in a small negative reward. If the agent reaches the *goal* quickly and minimises the total torque of its joints, it will obtain a score of at least $+300$. If at any point the agent falls over, a reward of -100 is returned and the episode ends. This is represented by the reward function:

$$R(t) = \alpha(x(t) - x(t-1)) - \beta \sum_{i=0}^{3} |a_i| - 100 I\{y(t) < p_y\} \tag{3.3}$$

Where $(x(t), y(t))$ is the 2D position of the agent at time t, $|a_i|$ is the absolute value of the i^{th} action of the action vector, and $I\{y(t) < p_y\}$ is the identity value which is 1 when $y(t) < p_y$, otherwise 0. α and β are scaling parameters which we set to their default values: $\alpha = 4.33$, $\beta = 0.028$. p_y is set to the terrain height so that the termination penalty of -100 is applied when the agent's hull position collides with the terrain. The completion *goal* is defined as the agent's horizontal position $x(t)$ exceeding a value of 90. An *episode* is one sequence of {state, action, reward, state, ..., etc.} until the environment terminates from the agent falling over or the agent reaching the goal. The maximum number of time steps before the episode is terminated is 1600, but from experiments we found that this could be reduced to 1000 time steps to allow for faster training.

3.1.2 Bipedal Balancer

The *Bipedal Balancer* environment consists of the same four-jointed agent as in the *Bipedal Walker v2* environment. The agent starts at some initial random spawn point $p_0 = (x(0), y(0))$, and is controlled by four actions, $\mathbf{a} \in \mathbb{R}^4$, (one action for each torque joint) with an observation vector of 24 features, $\mathbf{s} \in \mathbb{R}^{24}$. We do not go into detail of the action and state vectors as they are analogous to the action and state representation in the *Bipedal Walker v2* environment (see Section 3.1.1). The reward function for the *Bipedal Balancer* environment is defined so as to encourage the agent to stay close to its initial position. An optimal agent must learn to balance on one leg. The reward function is defined as:

$$R(t) = 1 - L_1(\boldsymbol{p}(0), \boldsymbol{p}(t)) + \alpha \cdot I\{L(\boldsymbol{p}(0), \boldsymbol{p}(t)) > \beta\}$$

where $p(0) \in \mathbb{R}^2$ is the initial starting $(x(0), y(0))$ position of the agent, $p(t) \in \mathbb{R}^2$ is the $(x(t), y(t))$ position of the agent at time t, $L_1(\boldsymbol{x}, \boldsymbol{y})$ is the L1-distance between two vectors: $L_1(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{|x|} |x_i - y_i|$, $I\{\cdot\}$ is some identity function that is 1 when its contents '$\{\cdot\}$' are true and 0 otherwise, $0 < \beta \leq 1$ is the termination parameter, and $\alpha \leq 0$ is the scale of the termination penalty. As a default $\alpha = -5, \beta = 0.4$.

The *Bipedal Balancer* environment was used to unit-test implementations of the algorithms. Additionally, we wanted to choose a single policy-gradient algorithm to experiment with on the walking environment, and used the *Bipedal Balancer* environment to determine the best policy-gradient algorithm to use. The *Bipedal Balancer* environment was easier to test the algorithms with because it has a significantly shorter time horizon than the walking problem. To solve the *Bipedal Balancer* environment, the agent has to maintain constant torque on a 'standing' leg and then apply enough torque to move the non-'standing' leg behind itself to counter balance the weight of its hull's posterior. In contrast, the walking problem requires learning complex sub-policies, such as controlling the placement of a foot, in addition to dynamical balance and symmetry of leg movement.

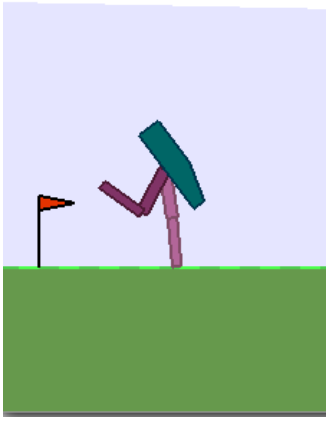


FIGURE 3.2: Screen-capture of the *Bipedal Balancer* environment.

3.2 How to Evaluate a Walk

To determine if the gaits learned by an agent are human-like we have to decide a quantitative, unbiased way to evaluate the walk. A natural way to evaluate the walk is to use an evaluation involving the reward, such as the *mean return*. But, the reward function does not necessarily associate with an effective gait. Additionally, we are interested in the gait becoming an emergent property of the environment and therefore want a simple reward function. We discuss the issues with using only the reward to evaluate a gait and then develop alternative methods of evaluation.

3.2.1 The Problem with Using Reward

The reward function for the *Bipedal Walker* environment (see Equation 3.3) does not tell our agent how to walk. Similar to how "a genie satisfies the letter of a request in an unanticipated way" [37], reward functions can be misinterpreted by their human creator and likewise exploited by the agent. *Exaptation* - using a feature for something other than its original purpose - is a common 'problem'. Incidentally, the problem with evaluating reward is well explained by *Goodhart's Law*: "When a measure becomes a target, it ceases to be a good measure" [70]. As will be demonstrated in the experiments of this report, the underlying intention of a vague reward function, such as the reward function we used, is usually exploited. This is especially true of evolutionary algorithms. This problem is akin to the use of training and testing data sets in machine learning. Without a testing data set, a learning model will exploit factors of the training data that were not the intended result. By analogy, it is therefore important that we evaluate an agent's learned gait via a process that was not involved in its learning.

3.2.2 Alternative Evaluation Methods

Here we outline an approach to evaluating a bipedal walk quantitatively via three methods: *robustness*, *stead-state stability*, and *phase-shift symmetry*. Unlike something like image classification, there is no 'correct' way to evaluate the quality of a walk. Some papers use qualitative methods and others use various quantitative methods. Because no one quantitative method can tell us the quality of the entire gait, we combine three different methods to try and get a global representation of the quality of the score.

1. *Robustness:* The robustness quantifies how well the agent performs in unseen environments. Because the agent cannot be expected to generalise to objects it has not been trained on, the new environment has to be similar to the environments it has been trained on. One way we can change the terrain and make it more difficult is by altering how the *velocity* of the terrain changes. Equation 3.1 describes how the terrain changes at each time step. For $a_g > 1$, the terrain position accelerates faster and is more likely to produce steeper surfaces. When $a_g = 0$ the terrain is completely flat.

   The agent is *only* trained on terrains where $a_g = 1$. To test for robustness, we run the agent on terrains of values $a_g \in \Omega = [1.0, 1.5, 2.0, 2.5, 3.0]$ and take an average of the mean return received over $N = 40$ runs for each value of s. To allow for quantitative comparison between models, we do a scaled weighted-sum of the average mean return for each acceleration value such that we define the *robustness value* as:

   $$\frac{1}{300} \sum_{a \in \Omega} [\frac{1}{N} \sum_{x=1}^{N} r_x(a)] \cdot a$$

   where $r_x(a)$ is the mean return of the agent when run on an environment with acceleration scalar a. We arbitrarily scale the mean return by dividing by 300 which is approximately the maximum mean return. Note that a high average mean return for a terrain of $a_g = 3$ contributes $3\times$ as much to the robustness score as a high average mean return for a terrain of $a_g = 1$. Figure 3.3 shows examples of terrains with different acceleration values.

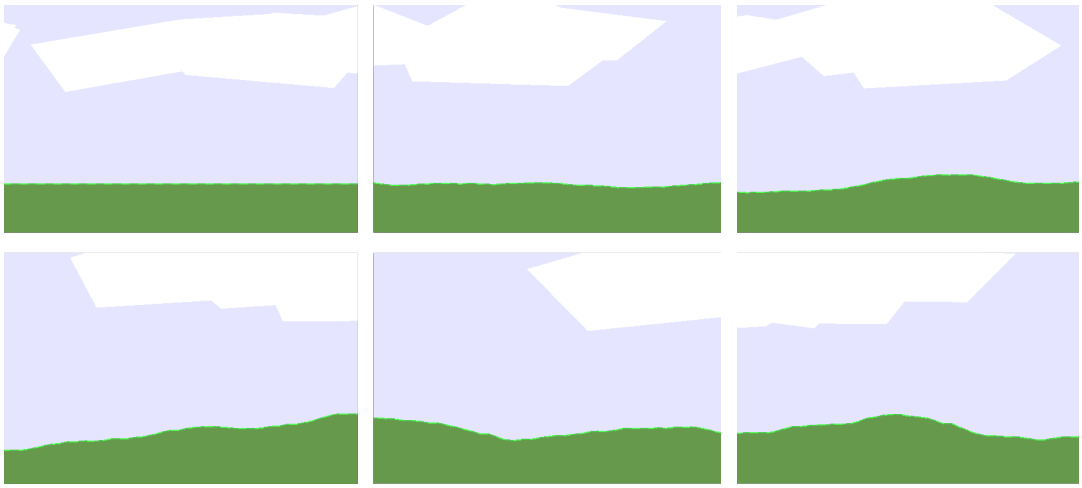


FIGURE 3.3: From left to right, top to bottom: terrains with acceleration values of $0.0, 1.0, 1.5, 2.0, 2.5, 3.0$.

2. *Steady-State Stability:* An important test for a natural gait is that, on a flat terrain, the agent's body should experience the same dynamics when taking the same step at different time intervals. For example, placement of the left foot should cause the agent's body to move in the same way every time the agent places that foot. The *phase* of a gait is defined as one full cycle of a leg which starts when a foot first breaks contact with the ground and ends when both feet have been replaced on the ground in the same relative position as they started (see Equation 3.5). For example, if the phase starts with back leg at the posterior of the agent and the front leg at the dorsal of the agent, the phase ends when the legs have returned to this same relative position. An example of phase is shown in Figure (REPLACE WITH CARTOON FIGURE) 3.4.

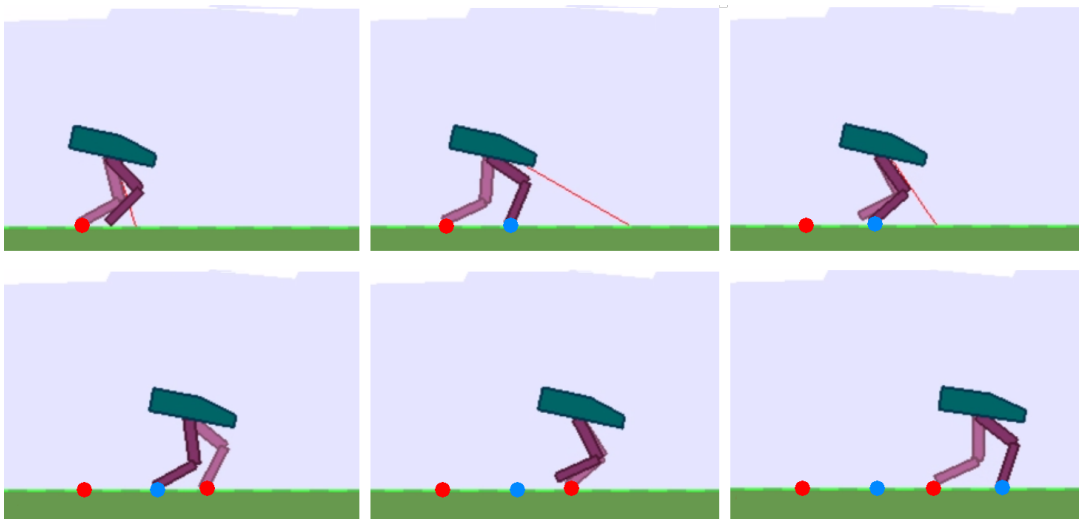


FIGURE 3.4: An example of a full phase is shown in frames 2-5. The first frame is *just* before a phase has started.

If, on average, a phase takes ϕ time steps and we define the *state* of an agent at time t as a vector of measurements $S(t) = (v_x(t), y(t), h_\theta(t))$, where $v_x(t) \in \mathbb{R}$ is the forward velocity of the agent, $y(t) \in \mathbb{R}$ is the vertical position of the agent, and $h_\theta(t)$ is the angle of elevation from the horizontal of the agent's hull

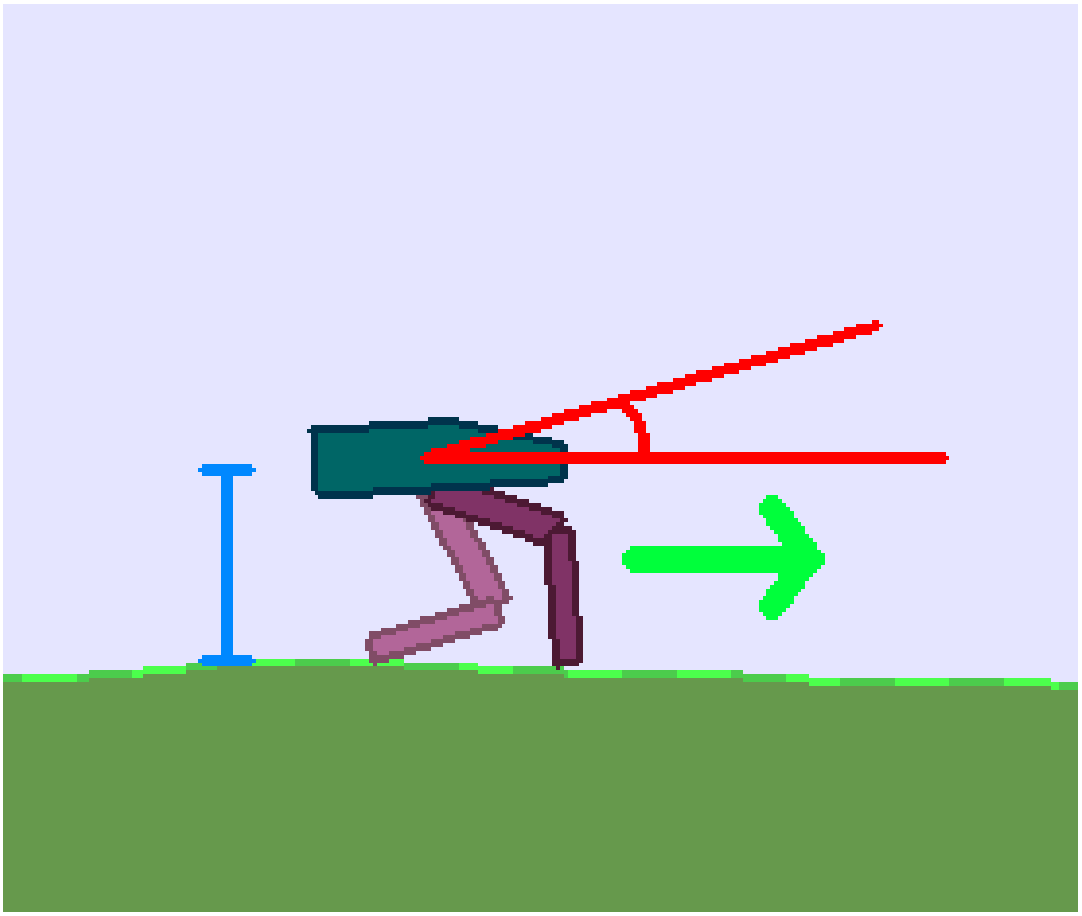


FIGURE 3.5: Red: angle of elevation, Blue: vertical position, Green: forward velocity

(see Figure 3.5), then the agent is in a *steady state* if the following holds:

$$\boldsymbol{S}(t) = \boldsymbol{S}(t + \phi)$$

That is, the state of the agent is the same between two phases. This is translated to an error metric over the entire *episode* of time length T such that:

$$\boldsymbol{E}(t : T) = (\boldsymbol{S}(t), \boldsymbol{S}(t+1), ..., \boldsymbol{S}(T))$$
$$E_{ss} = L(\boldsymbol{E}(t : (T - \frac{\phi}{2})), \boldsymbol{E}((t + \frac{\phi}{2}) : T))$$

Where $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ is some distance metric between two vectors $\hat{\boldsymbol{y}}, \boldsymbol{y}$. We use the *normalised root mean-squared error*:

$$L(\hat{\boldsymbol{y}}, \boldsymbol{y}) = NRMSE(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \sqrt{\frac{\sum_i (\hat{y}_i - y_i)^2}{\sum_i (\bar{y} - y_i)^2}} \tag{3.4}$$

Where $\bar{y}$ is the mean over all values in $\boldsymbol{y}$. The *NRMSE* is a natural choice for comparing the two states because the state of an agent is a time-series where the range of values differ across all agents. *NRMSE* is scale-invariant and suits time-series data well. A *NRMSE* of 1 is equivalent to predicting the mean of the dataset. Finally, to calculate the *steady-state stability value* as a qualitative error value, we run an agent on a terrain with zero acceleration ($a_g = 0$) and take the mean error over $N = 40$ runs:

$$\frac{1}{N} \sum_{i=1}^{N} \boldsymbol{E}_{ss}^{(i)}$$

The final error vector $\boldsymbol{E}_{ss}$ contains the errors for each of the *state* variables described earlier, that is:

$$\boldsymbol{E}_0 = error(v_x) \text{ (horizontal velocity SS-error)}$$
$$\boldsymbol{E}_1 = error(y) \text{ (vertical position SS-error)}$$
$$\boldsymbol{E}_2 = error(h_\theta) \text{ (hull angle SS-error)}$$

3. *Phase-Shift Symmetry:*

   The phase symmetry seems related to the steady-state stability, but it instead measures the symmetry *between the two legs* of the agent *within* a phase. The angle of the waist of one leg should be equal to the angle of the waist of other leg, with a constant phase offset such that:

$$\theta_w^b(t) = \theta_w^f(t + \frac{\phi}{2})$$

   where $\theta_w^b(t)$ is the angle of the joint of the back leg waist, $\theta_w^f(t)$ is the angle of the joint of the front leg waist, and $\frac{\phi}{2}$ is half the phase of the gait.

   We calculate the phase shift symmetry by computing the average phase shifts ϕ_b, ϕ_f for *each leg* from the step positions (see Equation 3.5). The phase shift

symmetry is then defined as:

$$\boldsymbol{\theta}^b_w(t:T) = (\theta^b_w(t), \theta^b_w(t+1), ..., \theta^b_w(T))$$
$$E_f = L(\boldsymbol{\theta}^b_w(t), \boldsymbol{\theta}^f_w(t - \frac{\phi_f}{2}))$$
$$E_b = L(\boldsymbol{\theta}^b_w(t + \frac{\phi_b}{2}), \boldsymbol{\theta}^f_w(t))$$
$$E_{ps} = \frac{1}{2}E_f + \frac{1}{2}E_b$$

Where for $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$ we use the *NRMSE* as before (see Equation). The last term is just taking an average of the error relative to the phase of the front leg and the error relative to the phase of the back leg. Again, to calculate the *phase-shift symmetry* as a qualitative error value, we run an agent on a terrain with zero acceleration ($a_g = 0$) and take the mean error over $N = 40$ runs:

$$\frac{1}{N}\sum_{i=1}^{N} E^{(i)}_{ps}$$

An example plot of the phase-shift symmetry is shown in Figure 3.6. The dark and light lines represent the angles of the back and front waist joints over time, respectively. The rectangles below the line indicate foot contact with the terrain (no rectangle means the foot is not in contact with the ground).

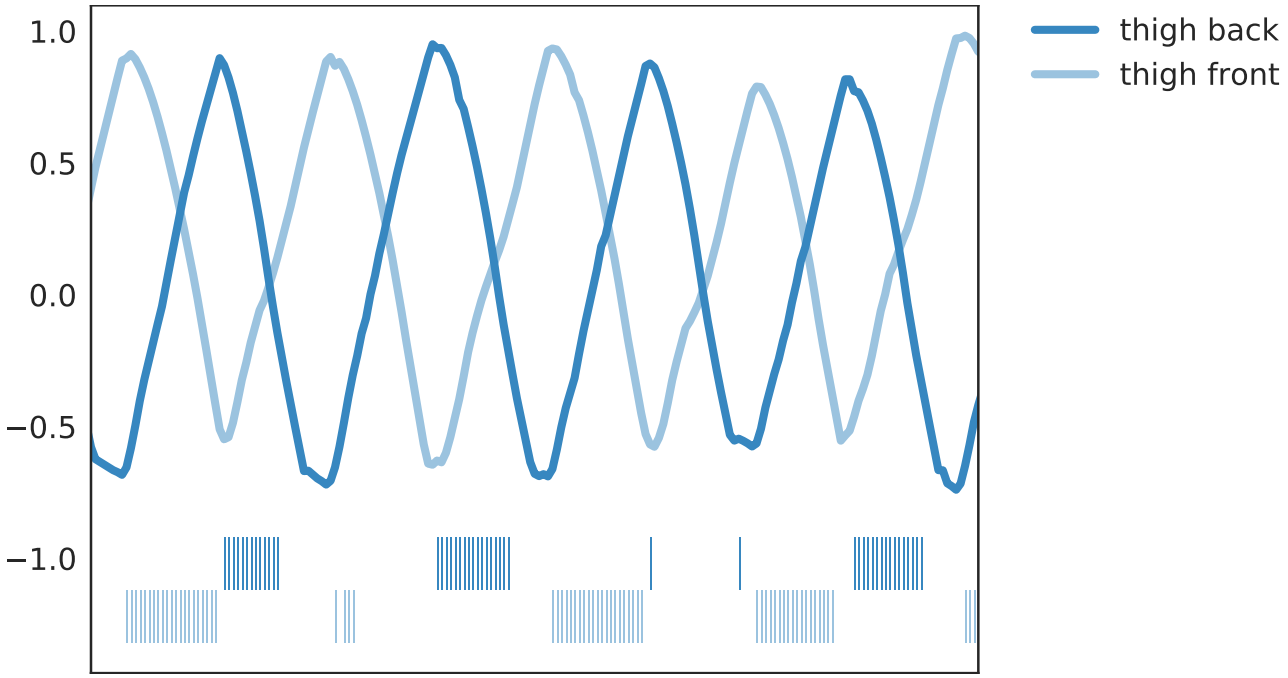


FIGURE 3.6: Example of a plot of phase-shift symmetry. The dark and light lines represent the angles of the back and front waist joints, respectively, over time. The rectangles below the line indicate foot contact with the terrain (no rectangle means the foot is not in contact with the ground). Because of small spaces in the contact rectangles, we smoothed out the contact values locally.

For both the *steady-state stability* and the *phase shift symmetry*, we have to compute the phase shift ϕ. This is done be finding all the time steps t where the foot *first* changes from no-contact to contact - we call these *first* contact points denoted by c_t - and then computing the average time difference from each contact point to its subsequent contact point:

$$\frac{1}{|C|-1}\sum_{c=2}^{|C|} c_i - c_{i-1} \tag{3.5}$$

where $C = \{c_1, ..., c_k\}$ is the set of *first* contact points.

Chapter 4

Evaluation Using Mean Return

It was crucial that our implemented algorithms were correct before we invested time training and experimenting with hyperparameters. There is recent research showing how deep-RL is unreliable for repeated trials using the same hyperparameters with different random seeds [53] [18] [26] [30] in addition to deep-RL software implementations being susceptible to errors [76]. Therefore it is sensible to unit test the implementations of the algorithms on simple environment where bugs can be easily detected early on.

This chapter presents initial experiments on the *Bipedal Balancer* environment, with some discussion on the training process for each algorithm. From the results, we discarded REINFORCE because it was difficult to train on the simple balancing problem, and we used DDPG for the walking problem. We then rigorously train DDPG, ES, and CE on the *Bipedal Walker* environment, exploring the effects of different hyperparameters. Fonally we present an in-depth analysis of various aspects of the DDPG and ES model architectures and how they could have affected the maximum *mean return* over time. The results show that although DDPG converged faster, ES achieved a higher maximum *mean reward*.

4.1 Results on the Bipedal Balancer Environment

In this section we present results of the policy gradient algorithms REINFORCE and DDPG, the evolutionary strategy algorithm, and the cross-entropy method algorithm on the *Bipedal Balancer* environment, and then briefly discuss whether the *Bipedal Balancer* environment was successful in helping unit test the algorithms. Comparison between algorithms is not a concern in this section - time was not consumed rigorously optimising the models for each algorithm. We concluded with choosing one policy gradient algorithm to use on the *Bipedal Walker* environment.

4.1.1 Implementation

We trained all algorithms for a maximum of $3,000$ episodes, with early-stopping once converged. For each algorithm we outline the architectures explored and the final ones used. Our exploration method was a simple grid search and does not necessarily reflect the optimal model. We are just interested in knowing that the implementations of our algorithms work.

REINFORCE

The REINFORCE algorithm required significantly more experimentation and adjustment than initially anticipated. This was mostly due to the diversity of implementations of the algorithm that used heuristics beyond the mathematical theory (REF). These heuristics are central to its success in different domains. Training REINFORCE was frustrating as we experimented with many architectures but rarely found success. We tried architectures of single-hidden layers of $\{16, 32, 64, 128\}$ units, batch sizes of $\{1, 5, 10, 25, 50, 128, 256\}$, with and without batch-normalisation in the hidden layer, with and without a baseline prediction for the mean return, and learning rates of $\{0.01, 0.001, 0.0001\}$ with both Adam and SGD. The 'best' model had a hidden layer size of 32, a batch size of 25, and used Adam with a learning rate of 0.001 and a batch normalisation layer.

Deep Deterministic Policy Gradients

We experimented with architectures of 70x100, 150x200, and 300x400 hidden units, which are all various scales of the hidden unit model used in the original DDPG paper [40], and grid searched over actor learning rates of $\{0.001, 0.0001\}$, critic learning rates of $\{0.01, 0.001\}$, and $\tau \in \{1.0, 0.1, 0.01, 0.001\}$. The best model had 300x400 hidden units, an actor learning rate of 0.001, a critic learning rate of 0.01, and $\tau = 0.1$. Reward scaling is important for DDPG [26] and from experimenting we found it necessary to scale the reward by 10.

Evolutionary Strategies

Because of the simplicity of the *Bipedal Balancer* problem, a single-hidden layer neural network is probably the most complex model required to solve the task. We ran tests on models with $\{16, 32, 64\}$ hidden units, with a standard population size of 100 and a noise scale (standard deviation of the Gaussian distribution) of $\sigma = 0.01$. We also experimented with using *linear* and *tanh* activation functions. The model that performed the best had 32 hidden units with *tanh* activation functions. We used stochastic gradient descent with a step size of $\alpha = 0.01$ and did *not* normalise inputs to the network.

Cross Entropy Methods

For CE we experimented with hidden units of $\{0, 16, 32\}$ with either a *linear* or *tanh* activation function. The inputs were *not* normalised. We used a population size of 100 and selected the top 10 individuals each optimisation step. Instead of the usual method to recalculate the variance from the re-sampled individuals, we just used a constant variance starting at 3.0 and decayed linearly by $\gamma = 0.99$ because this was more stable and provided consistently good performance. The update rule was:

$$\sigma^2(t) = max(\gamma\sigma^2(t-1), 0)$$

This is a common heuristic to use [29]. The best model architecture had *zero* hidden units and *linear* activation functions.

4.1.2 Discussion & Results

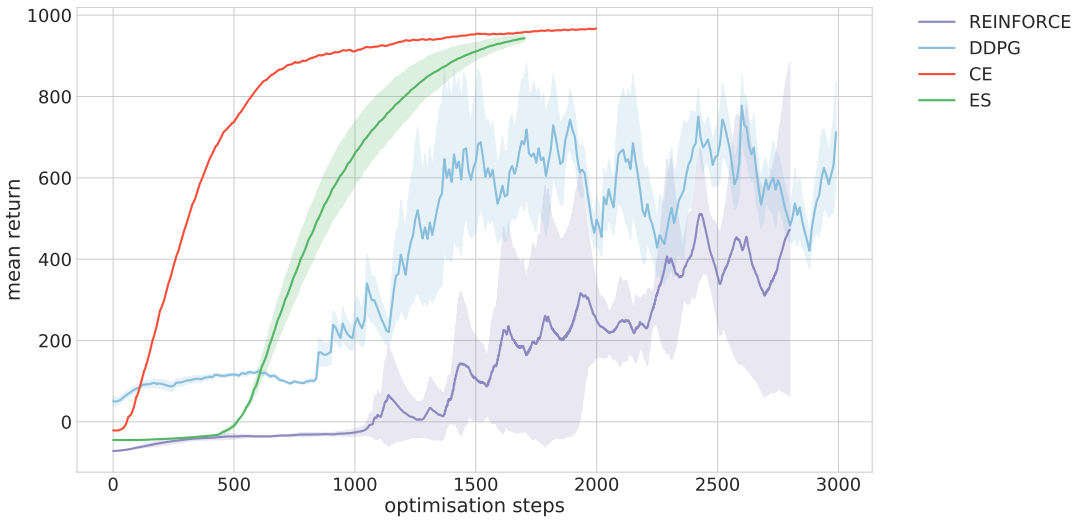


FIGURE 4.1: Results of the running ES, CE, DDPG, and REINFORCE on the *Bipedal Balance* environment.

From the results, we find that evolutionary strategies and cross entropy methods solved the task very efficiently as long as small models were used for the policy function. The REINFORCE algorithm did not guarantee convergence, and the algorithm was highly parameter sensitive. DDPG performed better than REINFORCE but did not converge. The reason behind DDPG not converging was probably due to using the *mean return* over time. During testing the model sometimes got a perfect score of ≈ 1000, but at other times it would completely fail and get a reward of < 200. This would hinder the calculation of the *mean return*. Of the two policy gradient methods tested, DDPG was the algorithm that produced good policies most consistently. REINFORCE took over 100 different experiments to try to find a good implementation. Therefore *we chose to use DDPG as our policy gradient algorithm* for the *Bipedal Walker* environment.

4.2 Results on the Bipedal Walker (v2) Environment

In this chapter we aimed to gain a rigorous understanding of the algorithms we are working with. From the *Bipedal Balancer* results, we found DDPG was the easiest to train of the two policy gradient methods used. We therefore chose to use DDPG as the policy gradient algorithm, and continued using ES and CE. The following sections outline the results for various hyperparameter settings and model architectures for training each algorithm on the *Bipedal Walker v2* environment, with the aim to gain an understanding of how the algorithms perform in relation to the commonly used maximum *mean return* metric. This chapter stands as a preliminary evaluation of the performance of the models and in the next chapter we show how the use of maximum *mean return* does not necessarily entail that a human-like walking policy has been learned.

4.2.1 Implementation

During the training of all algorithms, we utilise an 'early termination' process to reduce training time. Every d time steps, we check if the agent is making progress in its environment by determining if the error on the environment state after d time steps is below some threshold:

$$I\{\sum_{i=1}^{|\mathbf{s}|}(s_i(t) - s_i(t+d))^2 < \phi\}$$

Where $\mathbb{I}\{\cdot\}$ is an identity function such that if '$\cdot$' is true, the environment terminates. We set $d = 3$ and $\phi = 0.01$. Because this was a termination condition, there is the question of whether a reward penalty should be used - similar to the termination condition when the agent falls down. We tested the use of having no reward penalty and a -100 reward penalty and and found no difference in performance at train time. Therefore, we did not use a termination penalty. The 'early termination' is useful because it kills agents that get stuck and will not make progress - an example is shown in Figure 4.2.

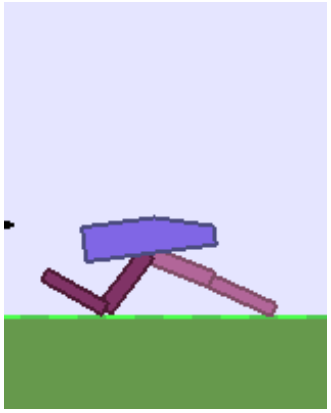


FIGURE 4.2: Screen capture of an agent early in training that is stuck in a state where it will not make progress because of its 'weak' policy.

Deep Deterministic Policy Gradients

As an initial baseline to improve from, we start with the same architecture used in the original DDPG paper [40] . The actor network consists of two hidden layers of 400 and 300 units, ReLU activation functions, with tanh activation functions on the final layer. The model is optimised using the Adam algorithm with a learning rate of 0.001. The architecture of the neural network is depicted in Section 2.1.2

Evolutionary Strategies

For ES, we started with a simple 32-unit hidden layer model , with a learning rate of 0.01, and noise value of $\sigma = 0.1$. The results on the walker environment were highly dependent on the population size, hence we did not make this value constant. An important parameter we introduced was the *maximum iterations*, which automatically terminated an environment after T timesteps. OpenAI used a similar heuristic [60]. This greatly helped to reduce training time and increase sample efficiency. The architecture of the ES model is the same as the *actor* network in Chapter 2, Section 2.1.2.

Cross Entropy Method

For the Cross Entropy Method, we started with a simple zero hidden-layer model and improved from there. Once again, manually annealing the variance parameter was more effective than re-computing the variance parameter. We found a variance of 10.0 annealed at a rate of $\gamma = 0.998$ worked well. For elitism population selection we used the heuristic of selecting the top 10% of the population. Similar to the ES training, we use *maximum iterations* to terminate the environment early.

4.2.2 Results: Deep Deterministic Policy Gradients

There are various improvements and suggested tweaks for DDPG that can either help or hinder the results depending on the context of the problem [49] [40] [10]. Here we provide a more in-depth exploration of the experiments we performed, using the *mean return* over time as the performance metric.

Reward Scaling

The DDPG algorithm is not invariant to the scale of the reward. The scale of the reward directly affects the value of the predicted return learned by the critic network because larger return values result in larger gradients propagated through the actor network. Reward scaling has been shown to have a large effect across various environments [26] [18] [22]. We explore scales of $\times 0.1, \times 1.0, \times 10.0,$ and $\times$ 100.0 in addition to also removing the termination reward. The results are shown in Figure 4.3.

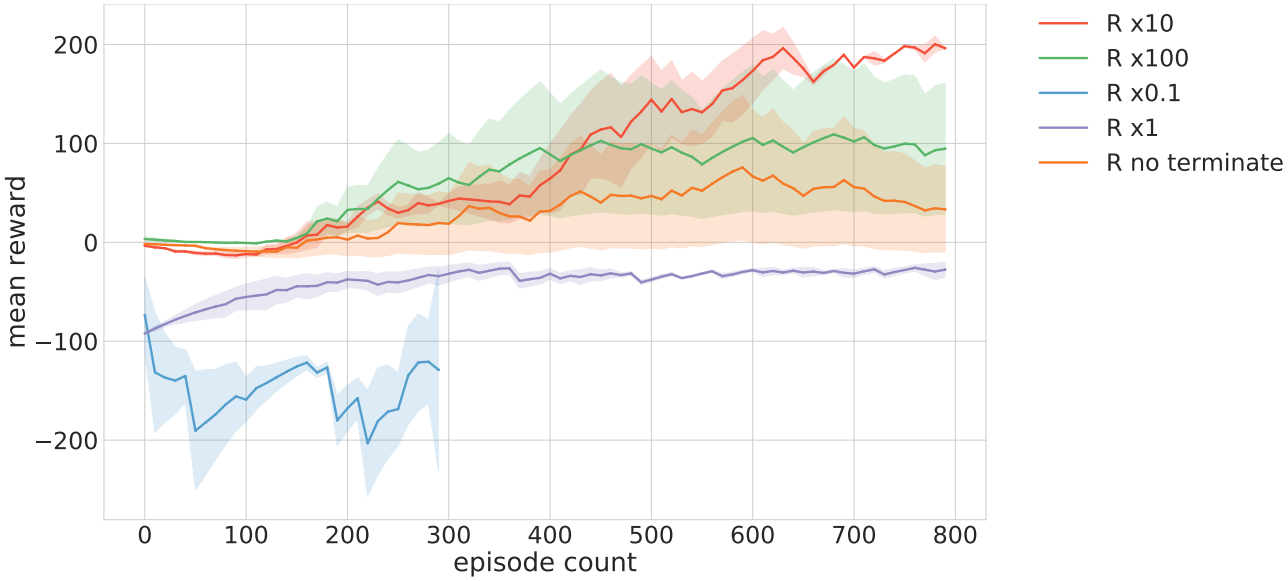


FIGURE 4.3: Experiments with different reward scales of $\times 0.1, \times 1.0, \times 10.0,$ and $\times$ 100.0. Also a reward scale of $\times 1.0$ with the termination reward removed.

Scaling the reward by $\times 10.0$ produces the most stable reward improvement over time. We include this .0 reward scale for all future experiments.

Small vs. Large Architecture

We explore how the model performance scales with the size of the hidden layers in the network. Specifically, we test scales of size $\times 1.0, \times 0.6, \times 0.25$, meaning we scale the number of hidden units in the layers by a constant value but maintain the ratio between the two layers. The results are shown in Figure 4.4.

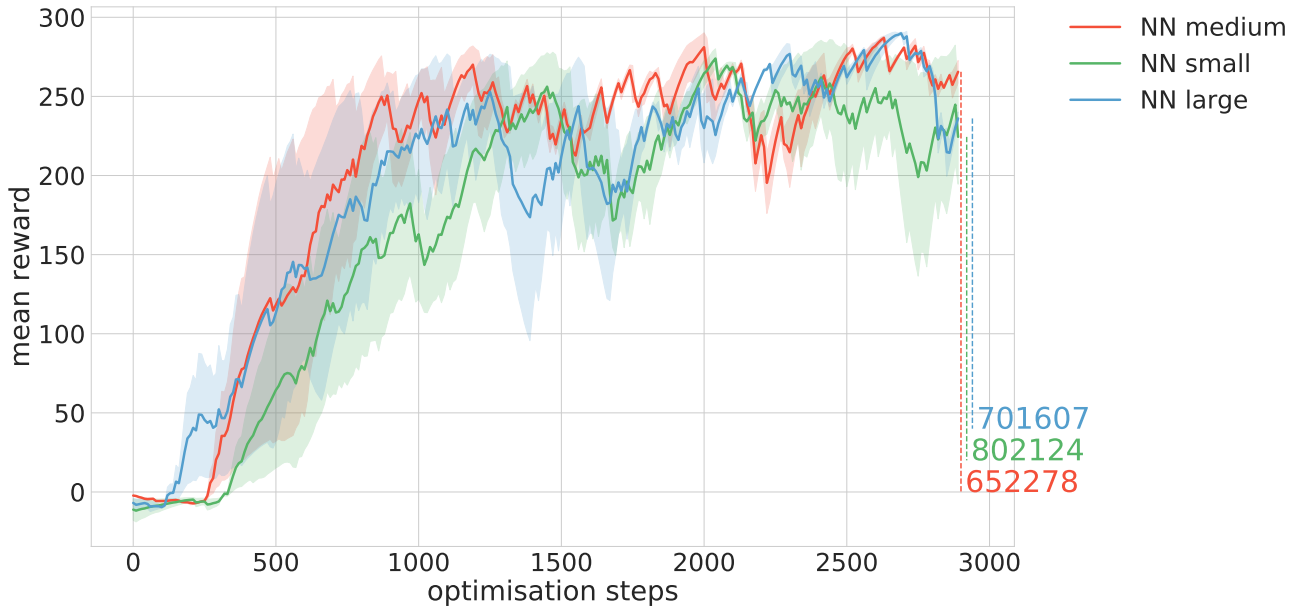


FIGURE 4.4: Comparison of three architecture sizes for the neural network. The end of the graph is annotated by the total number of iterations at the final epoch.

Gaussian Noise vs. O.U. Noise

During exploration, it is common to apply general Gaussian noise when working in a continuous action space [72]. The DDPG paper recommends Ornstein-Uhlenbeck (O.U.) noise [77] as an alternative [40]. O.U. noise works well for systems with inertia by correlating noise through time by using Brownian motion as a model [40]. As Paweł Wawrzynski puts it, O.U. noise prevents robots from "their [own] destruction" [79]. We test whether the use of O.U. noise improves learning versus Gaussian noise. The results are shown in Figure 4.5.

It is clear that O.U. noise produces faster convergence. This is due to exploration via O.U. being less jerky and more natural to robot dynamics [79], resulting in action sequences that the agent is more likely to perform with a good policy.

Target NN vs. Direct Training

Policy Gradient methods are infamous for their high variance [72]. The AC model helps reduce some of this variance [72], but it can be further reduced by introducing a target network [40]. Initially we found the recommended $\tau = 0.001$ did not work and used $\tau = 0.01$. We now explore whether this target network is necessary. When removing the target network, the learning rate must be scaled by τ's

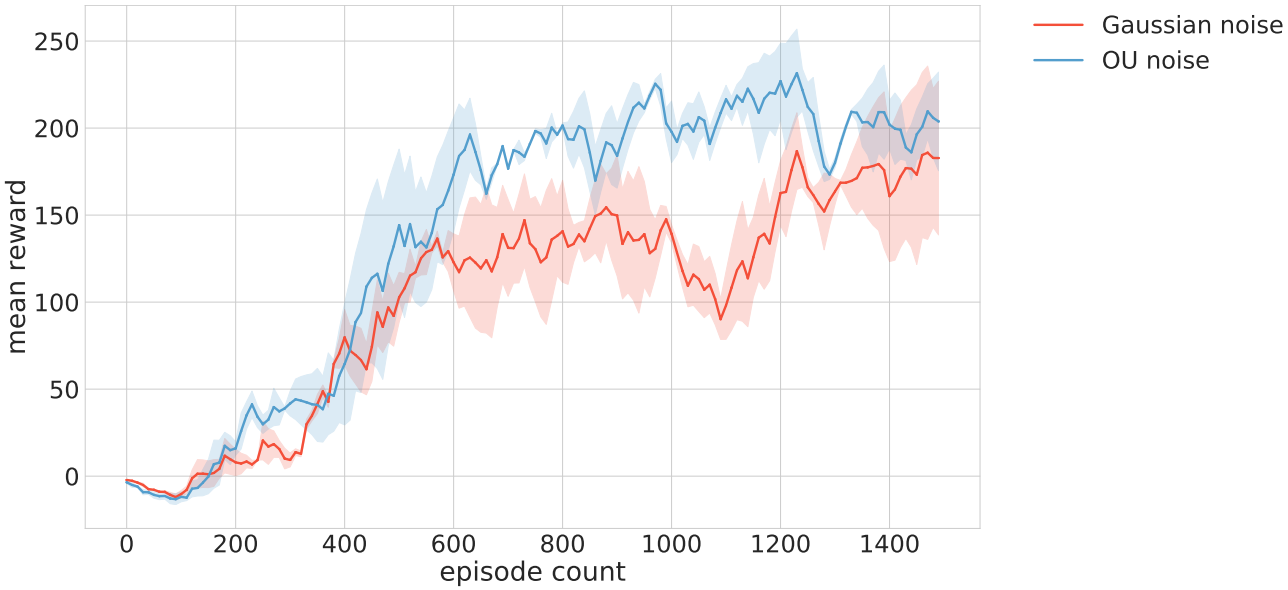


FIGURE 4.5: Results of using Ornstein Uhlenbeck noise versus Gaussian noise.

original value, otherwise the gradients will be too large to learn well. The results are shown in Figure 4.6.

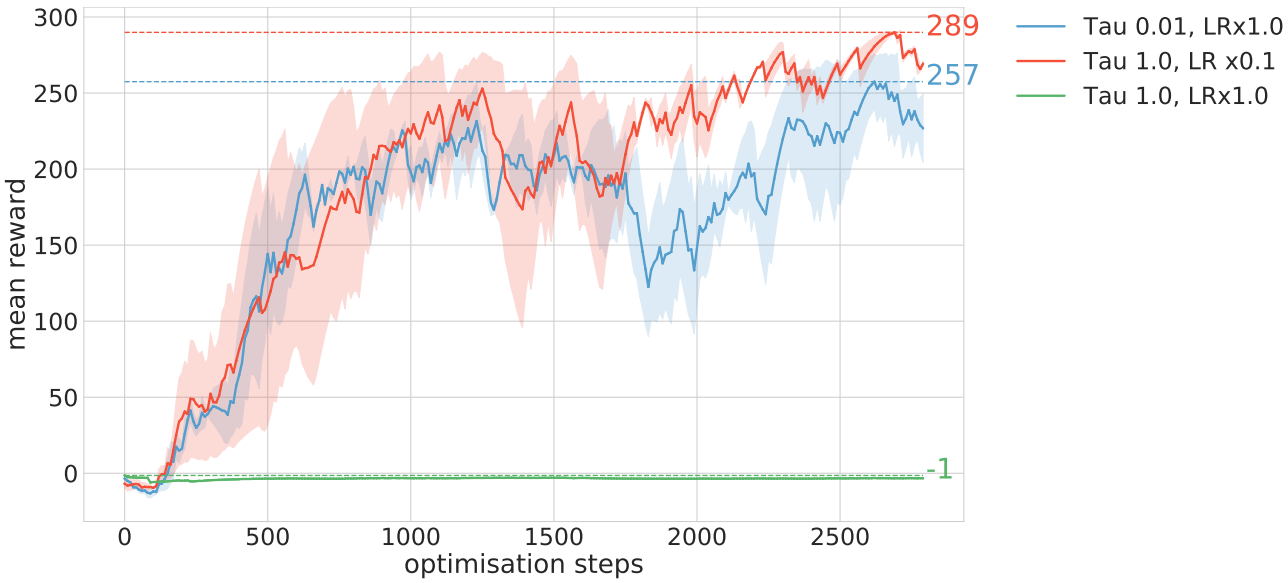


FIGURE 4.6: Results of values of τ. A τ of 1.0 (no target network) produces the best performance if the learning rate is scaled accordingly.

When scaling the learning rate by 0.1, we found that $\tau = 1.0$, and hence no target network, performed the best.

4 Batches per Iteration vs. 1 Batch per Iteration

One suggested enhancement is to quadruple the number of training steps per episode [10], effectively performing four gradient updates per time step in the environment instead of one. In each training step a new minibatch is sampled, therefore

increasing the number of minibatches per training step should speed up convergence and be more sample efficient because less interactions with the environment are required [10]. Results are shown in Figure 4.7. Increasing to four batches per

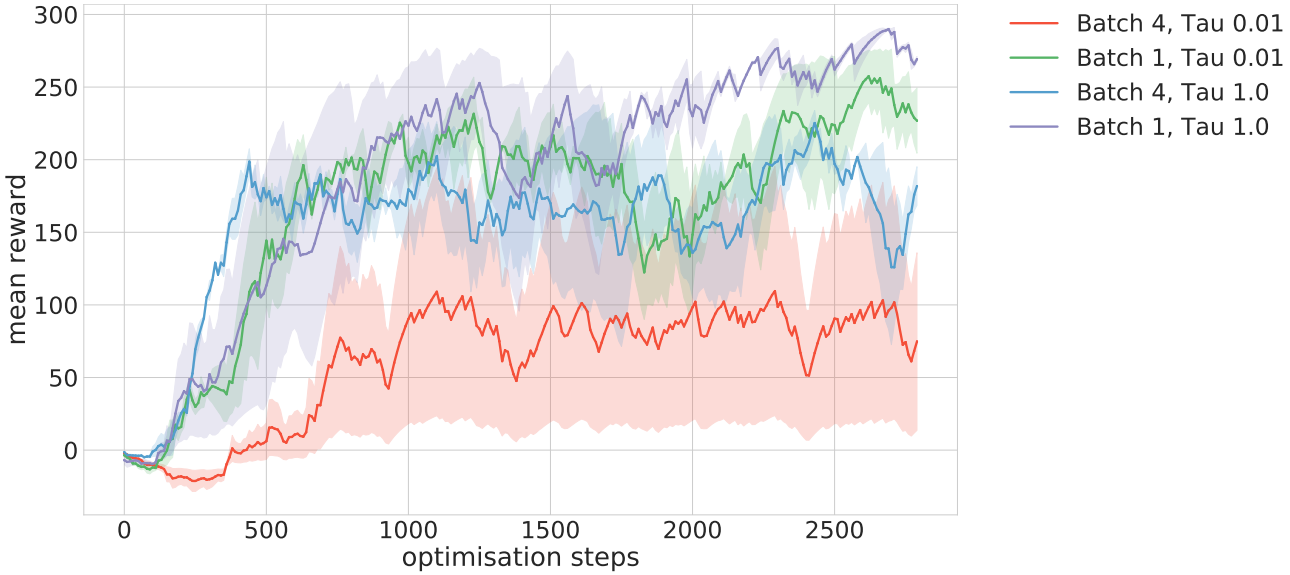


FIGURE 4.7: Training with four batches per training step versus one batch per training step.

training step does increase the speed of convergence, but the model quickly gets stuck at a local optima. We therefore retain the one batch per training step.

4.2.3 Results: Evolutionary Strategies

We explored how various hyperparameters for the ES algorithm as well as how the architecture of feed-forward neural networks affect the results performance. All results are compared using the *mean return* over time. Specifically, we explored how the population size and magnitude of the noise affect learning. Because ES is a gradient descent optimisation algorithm, we also explored how using Adam instead of stochastic gradient descent (SGD) helps improve performance. Lastly, we explored how the potentially higher number of dependent parameters for deeper neural network architectures affects performance.

Small vs. Medium vs. Large Population

The size of a population is a trade-off between accurate gradient approximation and convergence speed. A large population size will increase the gradient accuracy, but also increase the convergence time linearly [82]. The opposite is true for a small population size. In Figure 4.8 we show results for population sizes of 20, 40, and 80 when used with a 32-unit hidden layer model.

Small vs. Medium vs. Large Noise

The noise scale value σ is the standard deviation of the Gaussian distribution used to perturb the model parameters during gradient descent approximation. This

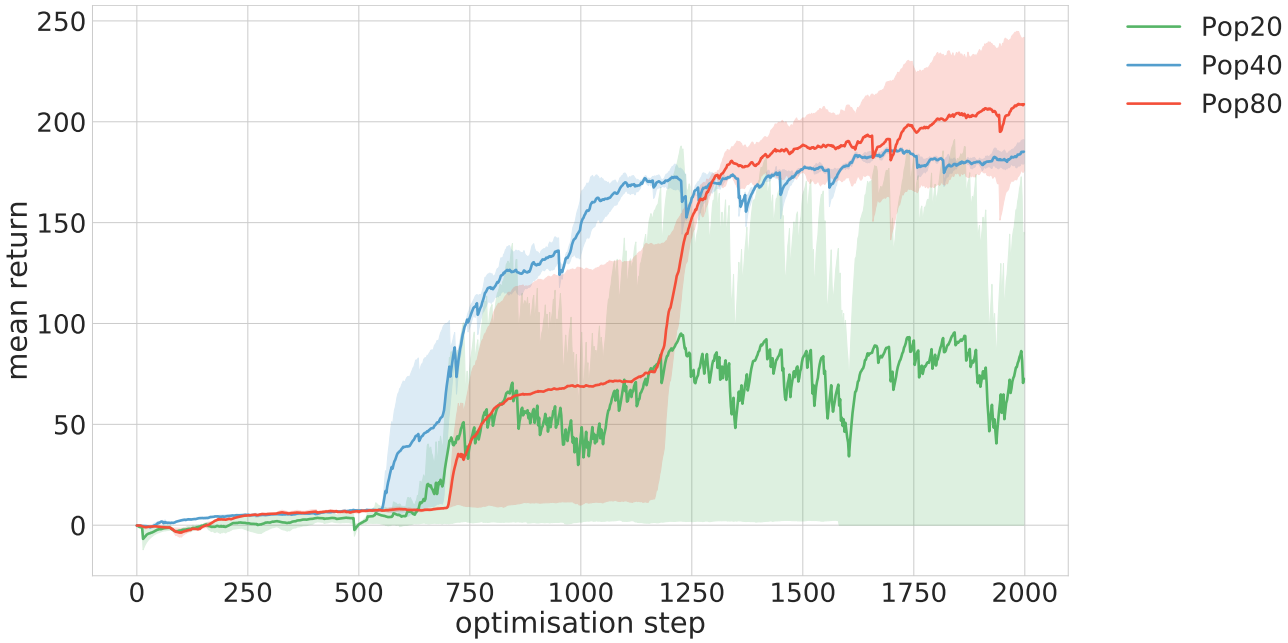


FIGURE 4.8: Comparison of using different population sizes for a model with 32 hidden units.

hyperparameter is crucial to ES learning well. Some papers recommend decaying the σ over time [82], and some state that it is not necessary [60]. We did not have time to explore this option and just kept a constant value throughout the whole training process. The results of a small experiment with σ are shown in Figure 4.9. From the

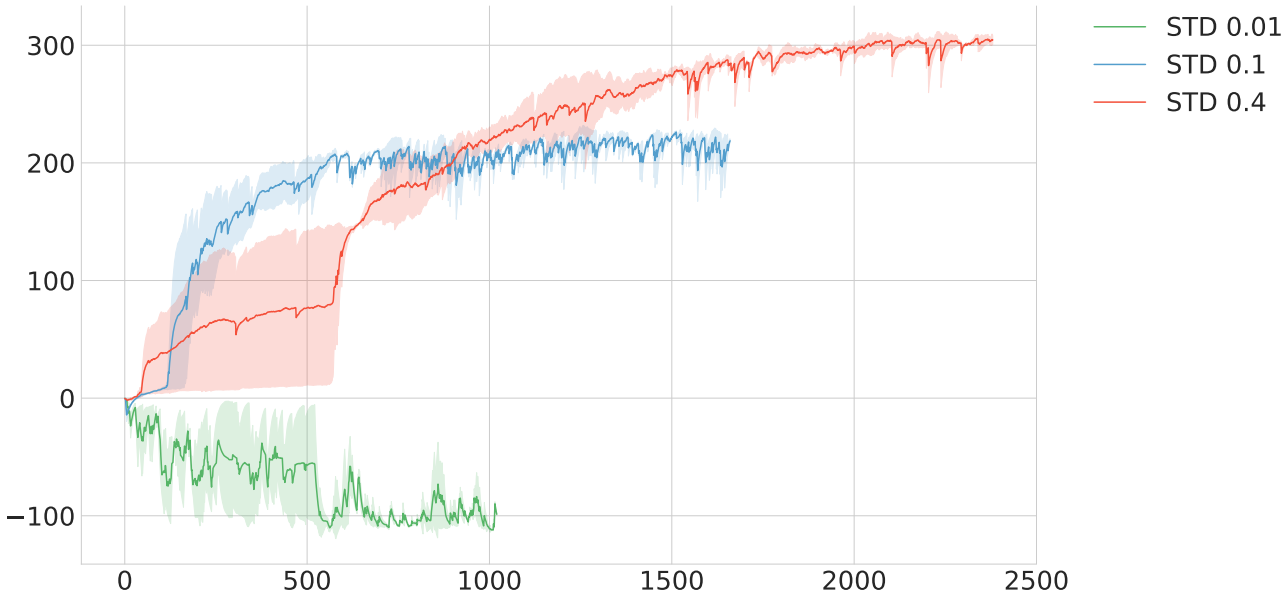


FIGURE 4.9: We experiment with a single 32-unit hidden layer, setting the noise value to $\sigma = 0.01, 0.1, 0.4$ each for a different models.

results, the noise parameter has a clear affect on convergence. If the noise parameter is too low ($\sigma = 0.01$) there is not enough exploration and the ES algorithm just oscillates around its initialisation position. If the noise parameter is too high ($\sigma = 0.4$) the ES algorithm will converge to local optima. This is because the perturbations are so large that the algorithm finds new models that are both better and worse and

on average it struggles to make further progress. When $\sigma = 0.1$, the ES algorithm smoothly ascended to a near-global optimum after an initial 500 time step period of noise.

Shallow vs. Deep Architecture

ES works well for neural-network optimisation because it can deal with optimising a large number of parameters. For example a single hidden layer 32-unit feed-forward neural network that we used has $24 \cdot 32 + 32 \cdot 4 = 896$ weight parameters. Because ES assumes no covariance relationship between parameters - unlike algorithms like CMA-ES, for example - this speeds up training specifically. But, we were interested to see how ES performed with deeper architectures where the dependence between parameters is different. For example, in the 32-unit architecture the adjustment of a parameter in the output weights is dependent on parameter adjustments of the input weights; in total there are $32 \cdot 4 = 128$ 'dependent' parameters according to this definition. But if we had a 2-hidden layer architecture with 19-units per layer such that it has the *same* number of parameters, we now have more 'dependent' parameters, $19 \cdot 19 + 19 \cdot 4 = 437$ in total. Because of this dependence, we hypothesise this model should be *harder* to optimise for ES. Additionally, we experiment with a 64-unit model to see how more parameters affects the optimisation. The results are shown in Figure 4.10.

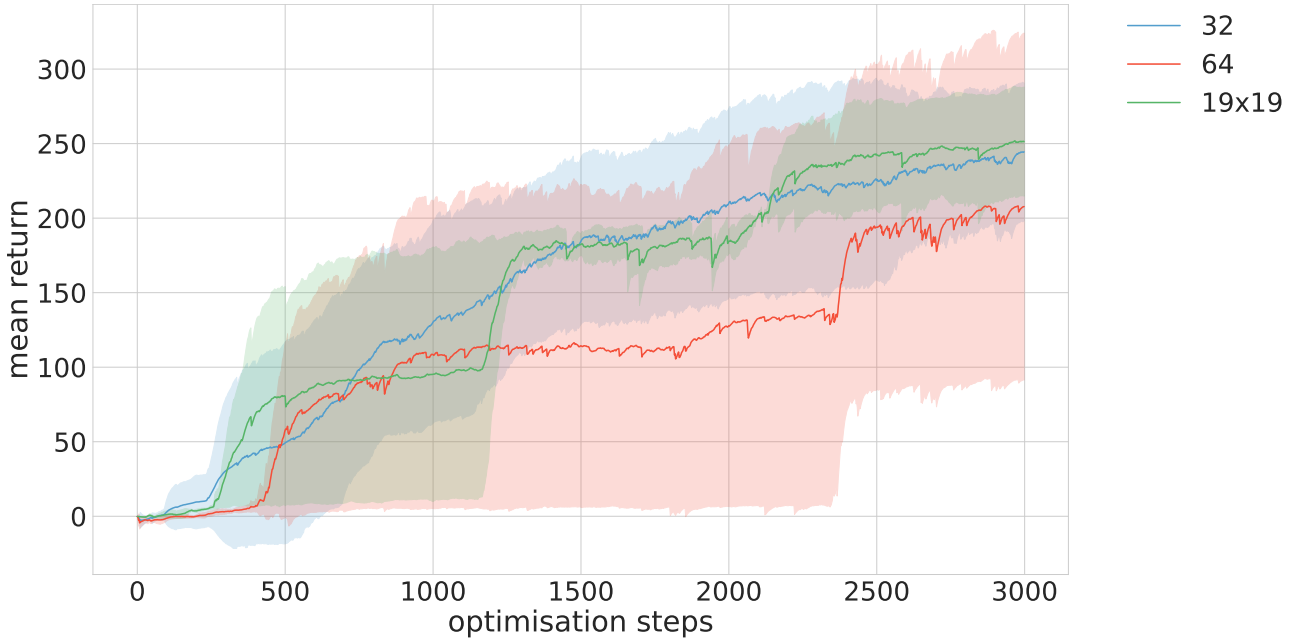


FIGURE 4.10: We compare architectures with 32 and 64 hidden units as well as an architecture with two hidden layers of 19 units each.

First, the results show that a smaller architecture of 32 hidden units is more stable than an architecture with 64 hidden units. This is most likely due to the reduction in dimensionality of the search space which ES has to perturb. Interestingly, the two-hidden layer architecture of 19 units was indifferent to a single hidden layer with 32 hidden units. Both architectures have the same number of parameters to be optimised, therefore this implies that architecture depth has less affect with ES

than hypothesised. Our analysis at the end of this section provides some justification as to why the 'dependence' of parameters for deeper architectures may not affect results. Essentially, ES does not seem to learn diverse policies that utilise this parameter inter-dependence.

Learning Rule

The Adam optimisation algorithm [35] is central to modern deep-learning performance. We explore how using Adam instead of SGD in the ES algorithm improves the training of the model. Specifically, we use a learning rate of 0.01, momentum of 0.9, and window rate of 0.999.

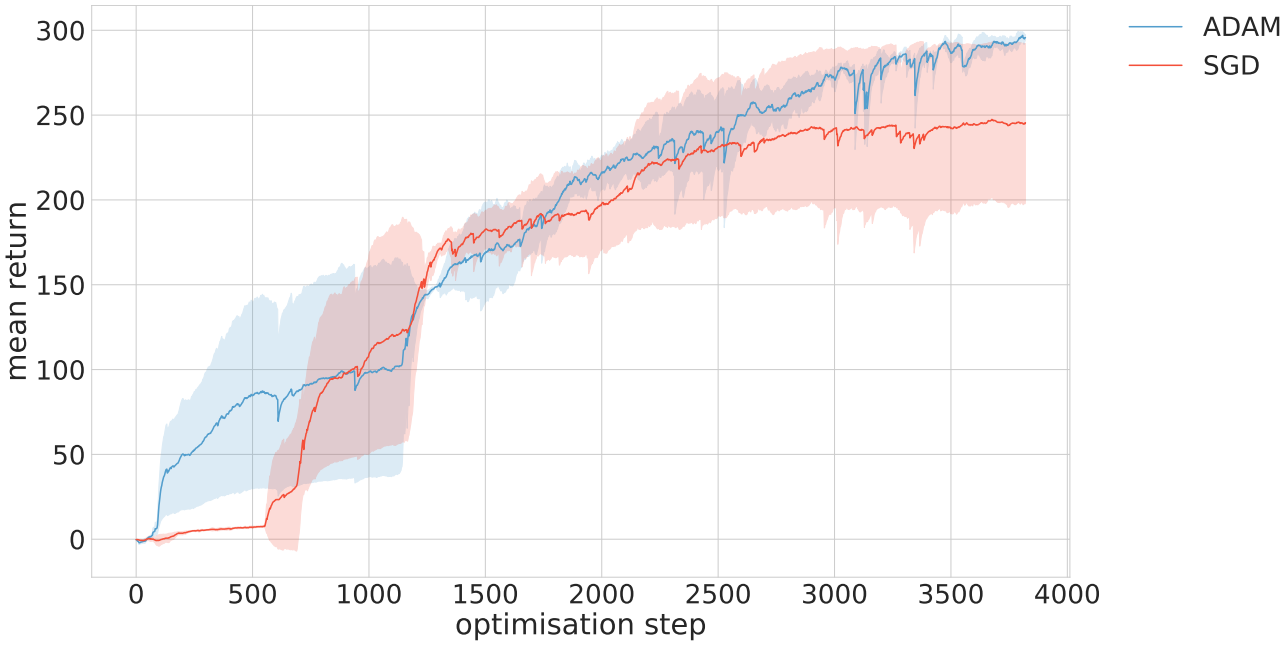


FIGURE 4.11: Comparison of Adam and SGD with a learning rate of 0.01. For Adam we used a momentum of 0.9 and a window rate of 0.999.

The use of Adam results in a gain in the maximum *mean reward* and converges to a *mean reward* of ≥ 310 versus the best SGD converging to a *mean reward* of ≤ 300. Additionally, Adam is more consistent with its convergence. We therefore recommend the use of ADAM over SGD when using ES.

4.2.4 Results: Cross Entropy Method

For the baseline algorithm, CE, we explored a range of hyperparameters and model architectures. The best model architecture had a single hidden-layer with 32-units and *tanh* activation functions. The results for CE were noisy; sometimes CE would produce an agent that achieved returns of > 140 and other times the agents would consistently fail. Still, for a continuous-action control task, CE performed better than anticipated and makes a suitable baseline model. In Figure 4.12 we plot the *mean return* of some of the models we explored. Note that models which produced converging low scores early on were cut-off.

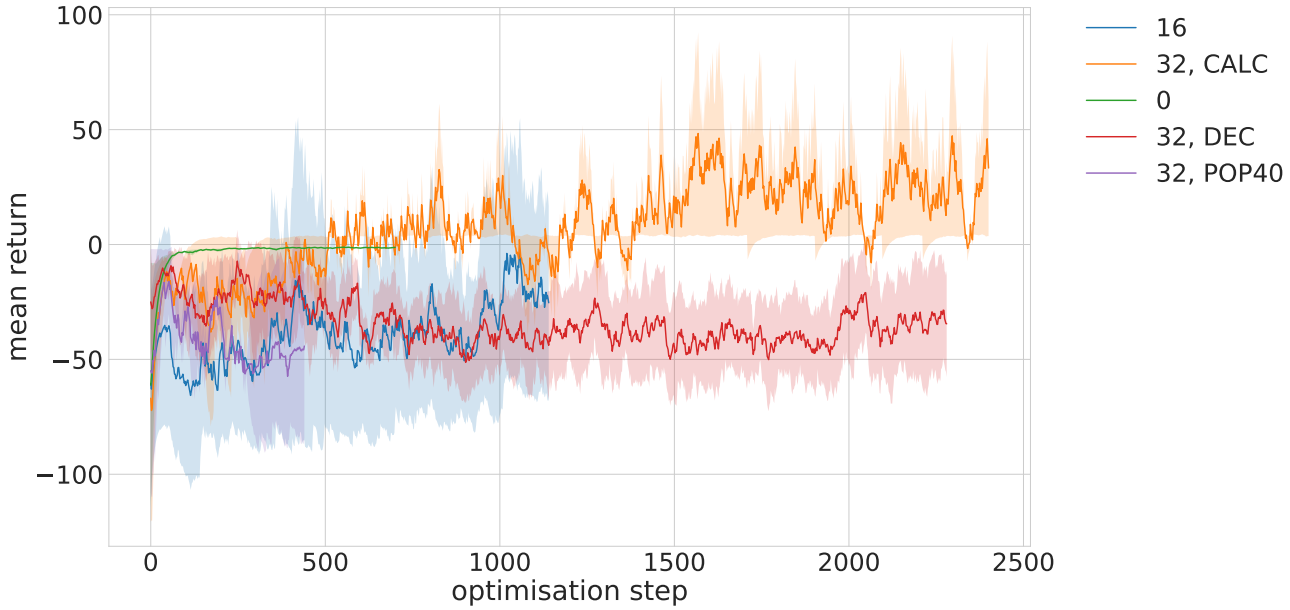


FIGURE 4.12: A sample of some the best CE results. 'CALC' refers to a model that 'calculates' the variance from the sample population. 'DEC' refers to the simple variance 'decay' process like the one used for the *Bipedal Balancer* environment. The number denotes the units in the hidden-layer. All models have a population size of 80 and the top 4 individuals are taken each round. 'POP40' refers to a model that used a population of 40 instead. We found that $\sigma = 10.0$ was a good starting variance.

4.2.5 Sample Efficiency of DDPG and ES

DDPG is extremely sample *efficient*, making it popular for continuous-action control tasks which require a large amount of samples to converge. After an initial exploration stage, the number of time steps per environment remains relatively constant. This is because the agent does not need to be run until termination each training step. Exploration is injected directly into the policy on-line because DDPG is an *off-policy* algorithm, allowing the agent to learn from small sequences of actions. ES is very sample *in-efficient* due to it having to run multiple agents within the same environment before taking a gradient step. To alleviate this problem, we cap the maximum number of iterations for various models. We found that 900 maximum iterations performed best. In Figure 4.13 we plot the number of iterations for each environment until termination, independent from the optimisation step, where one environment for ES counts as N concurrent environments for N individuals in a population.

4.2.6 Discussion

This section has allowed us to gain an intuition about how the hyperparameters of the various algorithms contributed to the success of the algorithms. Overall, it seems that DDPG works best for the Bipedal Walker environment with larger architectures, a reward scaling of $\times 10$, U.O. Noise, and no target network. ES works best for the *Bipedal Walker* environment with a single 32-unit hidden layer, Adam optimiser, and a large population of 80. CE works best with a hidden layer size of

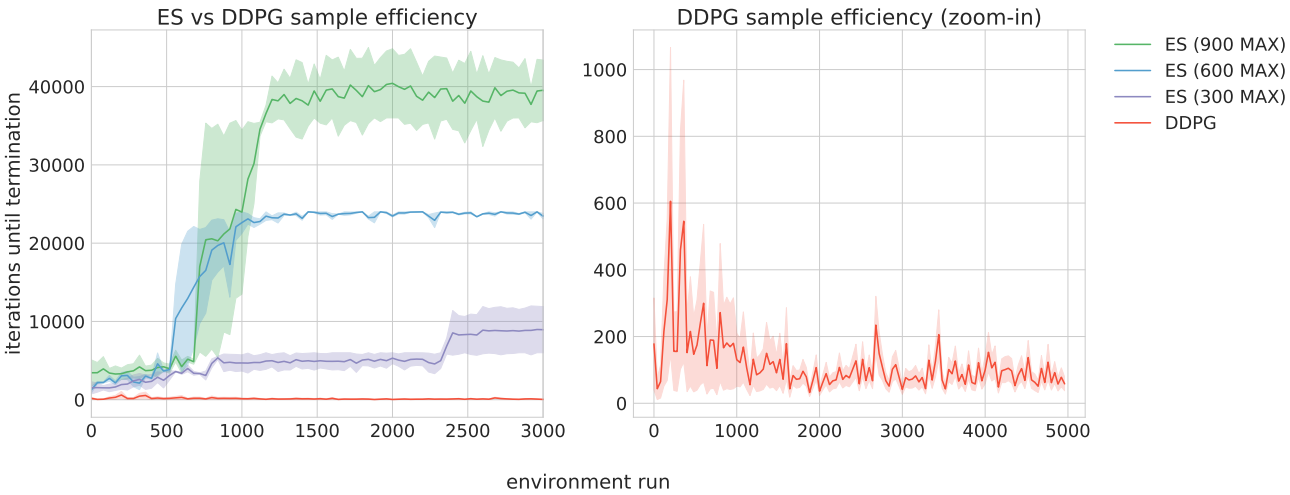


FIGURE 4.13: Sample efficiency for DDPG and ES. Left plot: ES with capped iterations of 300, 600, and 900, along with DDPG. Right plot: DDPG by itself. DDPG iterations start large because before learning to walk the agent can get stuck in a non-terminating position (see Figure 4.2).

16 units, *tanh* activation functions, a population size of 80, and an initial variance of $\sigma = 10.0$. The best model for each of the three algorithms is shown in Figure 4.14.

Both DDPG and ES outperform the baseline CE for this complex continuous-action control task. We can therefore conclude that DDPG and ES are effective learning algorithms for this problem. We are now interested in evaluating ES and DDPG on the basis of the learned walking policies. We do not evaluate CE because it did not get a *mean return* high enough to make it worthwhile to consider evaluation. At a low maximum *mean return* of 51, the best CE model would definitely not produce gait policies that competed with ES and DDPG. As an initial heuristic to choose which models to evaluate the walking policy on, we used the *mean return* over time, with early stopping once converged. Although the environment is considered "complete" with a mean return of ≥ 300 [48], we cannot guarantee that a good walking gait has been learned in this situation, as we demonstrate by results in Chapter 5. We therefore arbitrarily consider a ≥ 270 *mean return* as good enough to further evaluate its walking policy. In the next section we will look at the various policies learned by each algorithm among a range of models for each algorithm.

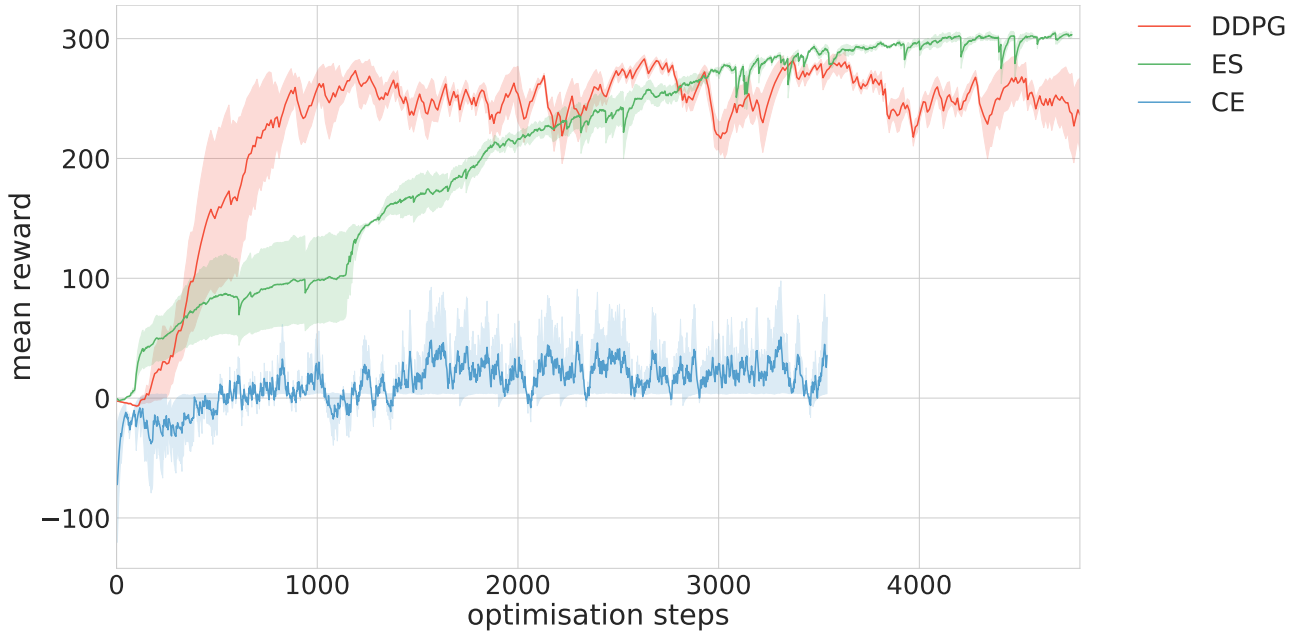


FIGURE 4.14: Comparison of the best model for ES, DDPG, and CE. The maximum *mean returns* are: ES: 307 ± 1.7, DDPG: 295 ± 3.3, CE: 51 ± 20.2

Chapter 5

Alternative Evaluation Methods for the Learned Gaits

In Chapter 3 we outlined three metrics for quantitatively evaluating a bipedal gait, and justified that *Goodhart's Law* meant we must have an evaluation that is independent of the maximum *mean return*. In this chapter we will now perform the walking evaluations on the models which obtained the highest maximum *mean reward* in addition to some sub-optimal models for comparison. Interestingly, we find that the maximum *mean return* does *not* directly correspond to learning a good walking gait.

In Figure 5.1 we show time-lapses of learned gaits that achieved > 290 maximum *mean return*. This shows that the 'best' gaits from Chapter 4 do not correspond to human-like walks. Therefore the maximum *mean return* evaluation was not effective for defining an optimal gait.

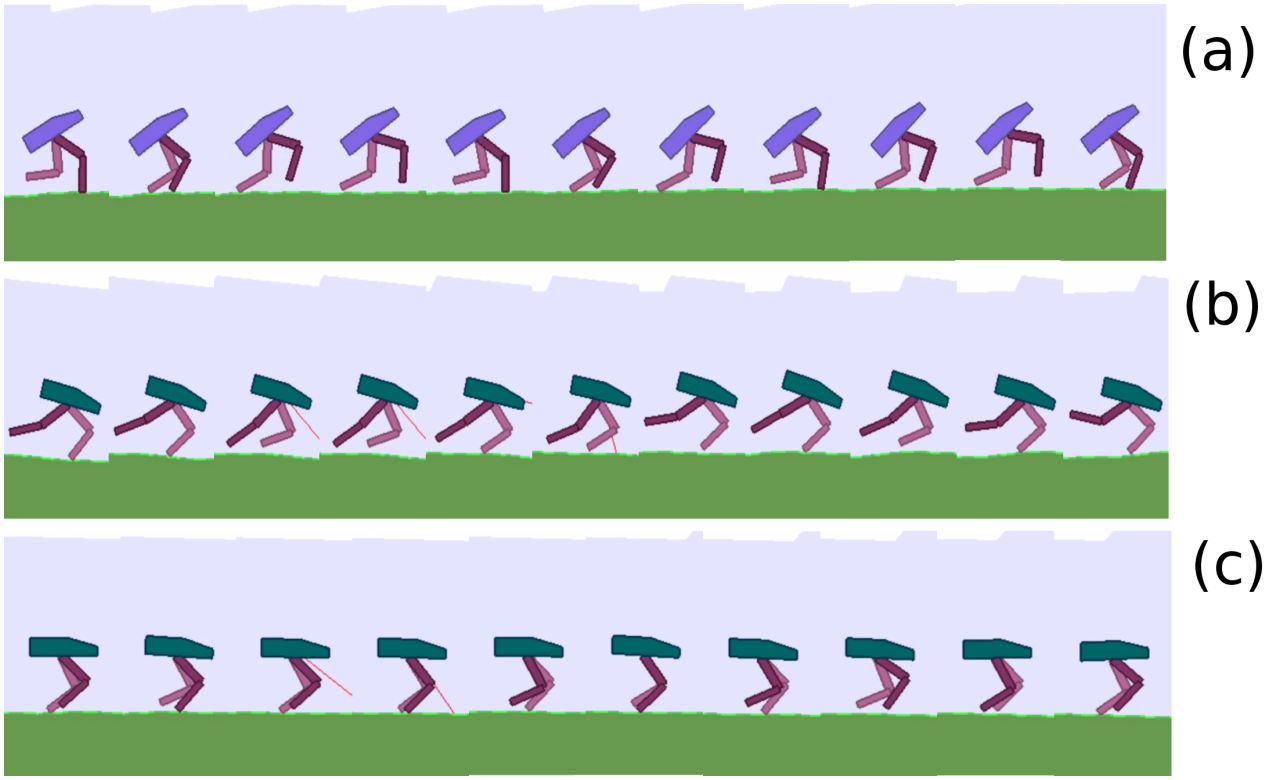


FIGURE 5.1: Time-lapses of three *very* different policies that all achieve a *mean return* > 290. (a) ES (b) DDPG #1 (c) DDPG #2

5.1 Evaluation Results

We evaluated five different models from the DDPG and ES algorithms. Collectively, the models have a variety hyperparameters for the algorithm they are from. Evaluating multiple models allowed us to show the general performance of the algorithm, independent of the hyperparameters chosen; this is especially important in reinforcement learning problems that are prone to 'lucky' seeds or initial starting conditions.

5.1.1 Deep Deterministic Policy Gradients

Results of robustness: DDPG performed well on the initial walking problem, which had a terrain of acceleration 1.0. When testing the algorithm's robustness, we found that it had a clear linear decrease in reward and was unable to move far when the velocity was 3.0. This result showed that DDPG strongly overfit its environment by learning a gait that performed well on simple terrain, but was not adaptable for harder, unseen terrain.

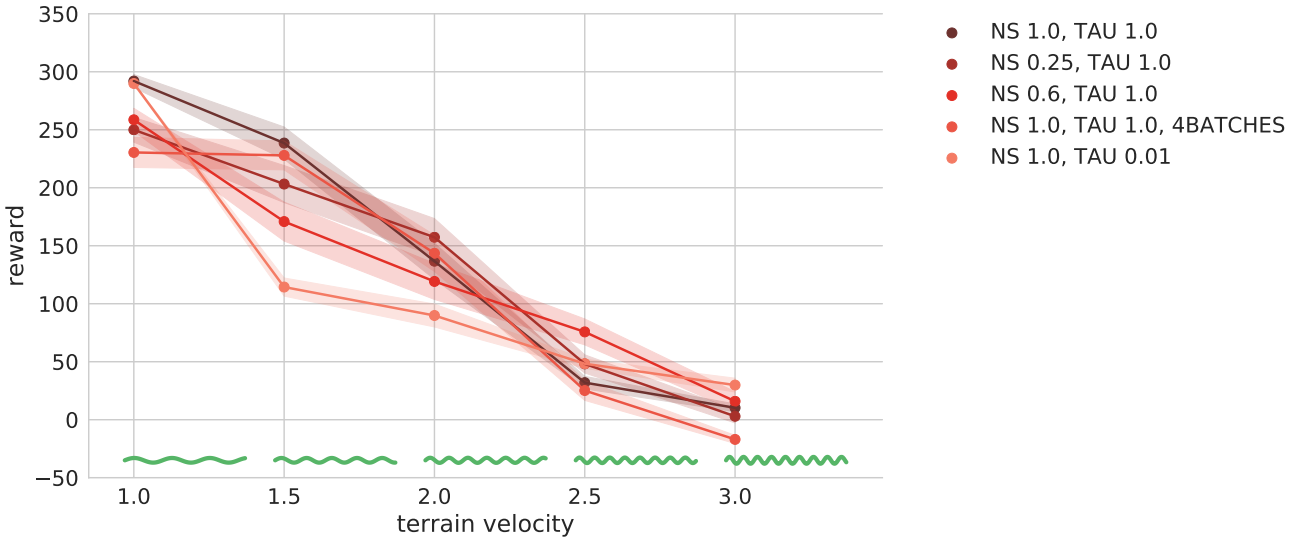


FIGURE 5.2: Robustness results for 5 different DDPG models.

Results of steady-state stability: The best models for DDPG scored well on the steady-state stability evaluation. This is because the algorithm was able to find symmetrical, sub-optimal walking gaits. Specifically, the model with a node-scale of 1.0 and $\tau = 1.0$ was the most stable. See Figure 5.3.

Results of phase-shift symmetry: DDPG produced gaits that were very symmetrical, as shown by the graph of the thigh angles in Figure 5.4. The trouble with the DDPG gait was the size of the phase was too small to be realistic - the best model skittered like an insect rather than walked like a human. This small phase size made the model unstable on inclines and explains why the model had low robustness. The interesting graph (b) in Figure 5.4 is the result of the model learning to hop on only one leg. Figure 5.5 shows the phase of the best walk (NS 0.25, TAU 1.0).

FIGURE 5.3: SS-stability results for 5 different DDPG models.

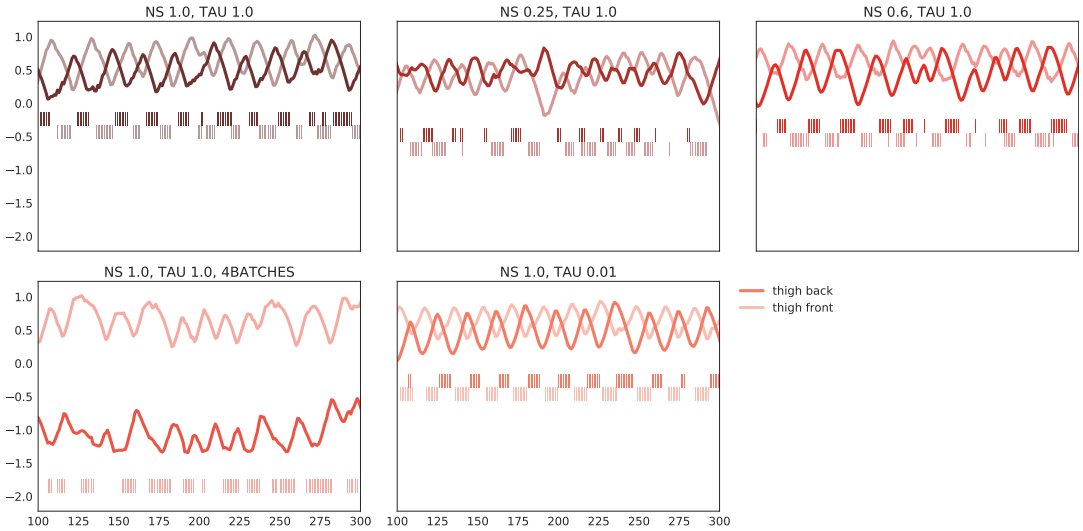


FIGURE 5.4: Phase-shift results for 5 different DDPG models.

MODEL			ROBUSTNESS	STABILITY			PHASE SHIFT
NS	TAU	BS		VERT.	PITCH.	VEL.	
1.0	1.0	1	**3.45**	1.44 ± 0.019	1.28 ± 0.016	1.34 ± 0.016	$\mathbf{0.44 \pm 0.0055}$
0.25	1.0	1	3.33	$\mathbf{1.12 \pm 0.22}$	$\mathbf{0.84 \pm 0.049}$	$\mathbf{1.19 \pm 0.057}$	0.63 ± 0.021
0.6	1.0	1	3.30	1.35 ± 0.018	1.23 ± 0.018	1.34 ± 0.017	0.61 ± 0.0047
1.0	1.0	4	2.90	1.96 ± 0.66	13.08 ± 11.66	1.55 ± 0.43	3.00 ± 0.041
1.0	0.01	1	2.84	1.52 ± 0.039	1.14 ± 0.030	$1.26 \pm 0.0.025$	0.60 ± 0.026

TABLE 5.1: Evaluation results for 5 different DDPG models.

5.1.2 Evolutionary Strategies

Results of robustness: The average model for ES is more robust than all DDPG models. Whereas the best DDPG model achieved a robustness of > 10 for a terrain velocity of 3.0, the best ES model achieved a robustness of > 150, which is $1,400\%$ better than DDPG. Observing the gait in Figure 5.9, it is clear the ES algorithm favoured a gait with a low centre of gravity and a wide stance, hence this explains the better performance of ES for the robustness metric. See Figure 5.6.

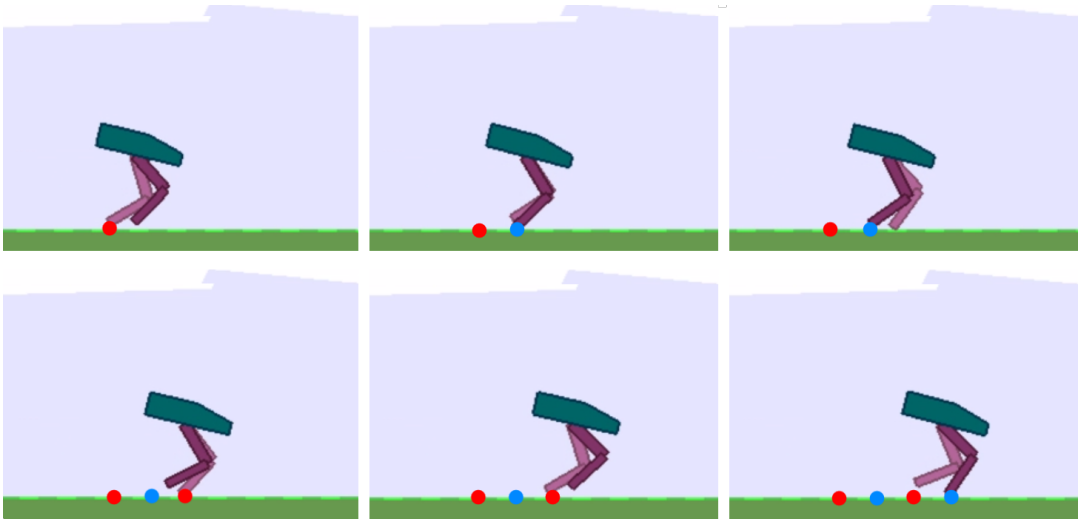


FIGURE 5.5: Time-lapse of the best DDPG model. Note how the legs overlap symmetrically, but placement of the legs are too close together.

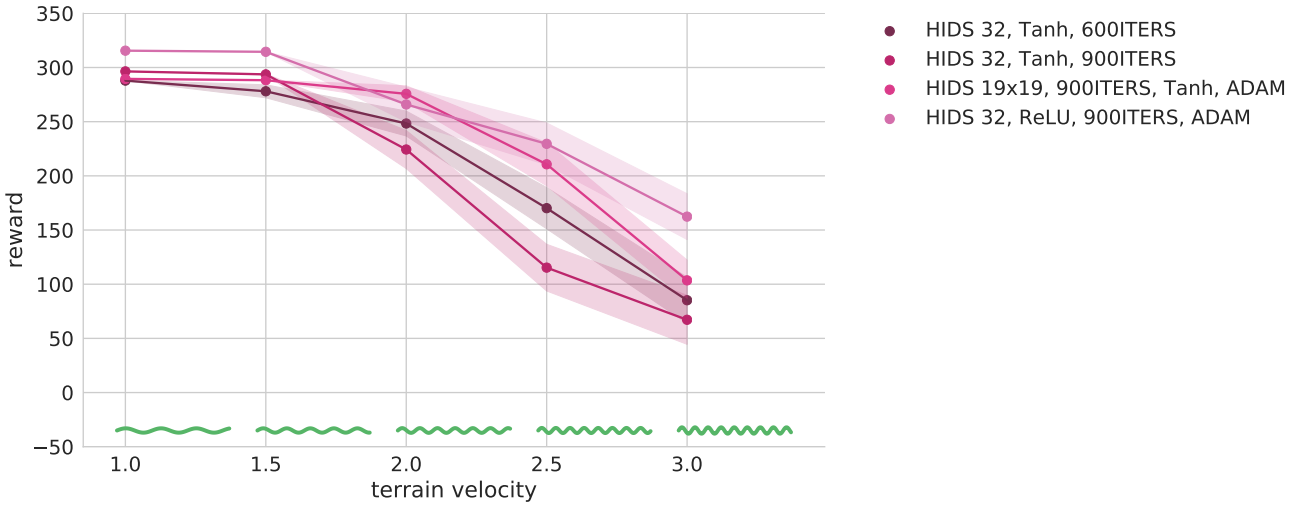


FIGURE 5.6: Robustness results for 4 different ES models.

Results of steady-state stability: The ES models also were more stable than the DDPG models. As a reminder, stability measures how consistent the dynamics of the agent are *between phases*. This means to score well, the agent has to be 'cyclic' in its movement. As we will discuss later, the ES model seems to favour minimum actions that are independent of the state space, hence it is easier for it to score well on this metric because actions are more independent of the terrain and therefore more likely to produce patterns. See Figure 5.7.

Results of phase-shift symmetry: It is clear this is where ES fails. The phase-shift symmetry is the most definitive of the metrics for determining if a gait is human-like. ES performs $\approx$ 800% worse than DDPG on this metric. The graphs in Figure 5.8, demonstrate how the ES models used only the back leg to push themselves from behind, while the front leg was minutely adjusted to support themselves when falling forward. Interestingly, the front waist joint is almost *never* moved in some models, which must be the result of a suboptimal model exploiting the torque penalty term in the reward function. See Figure 5.9 for a time-lapse of the best model.

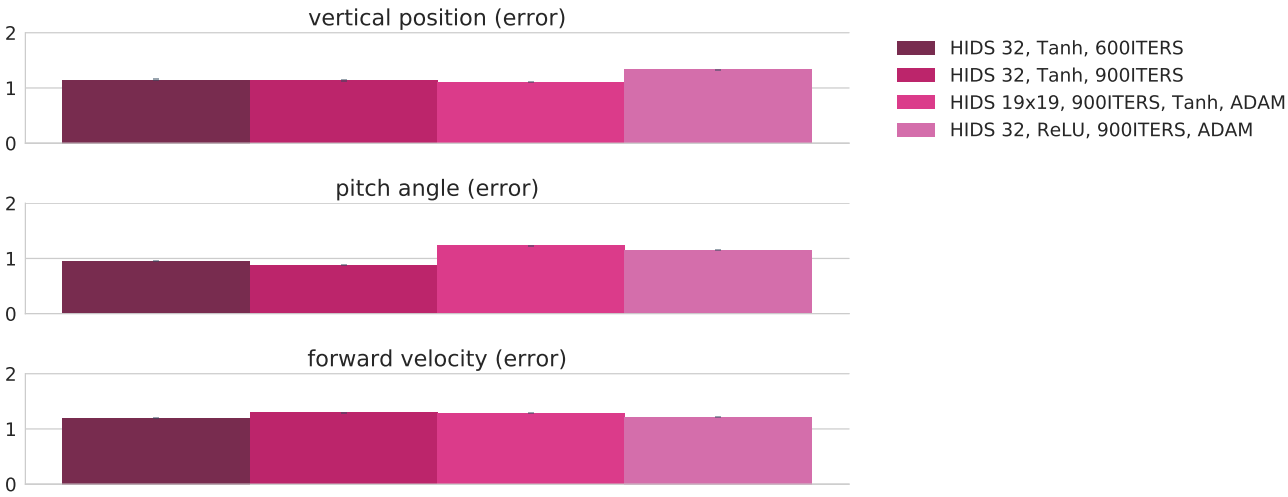


FIGURE 5.7: SS-stability results for 4 different ES models.

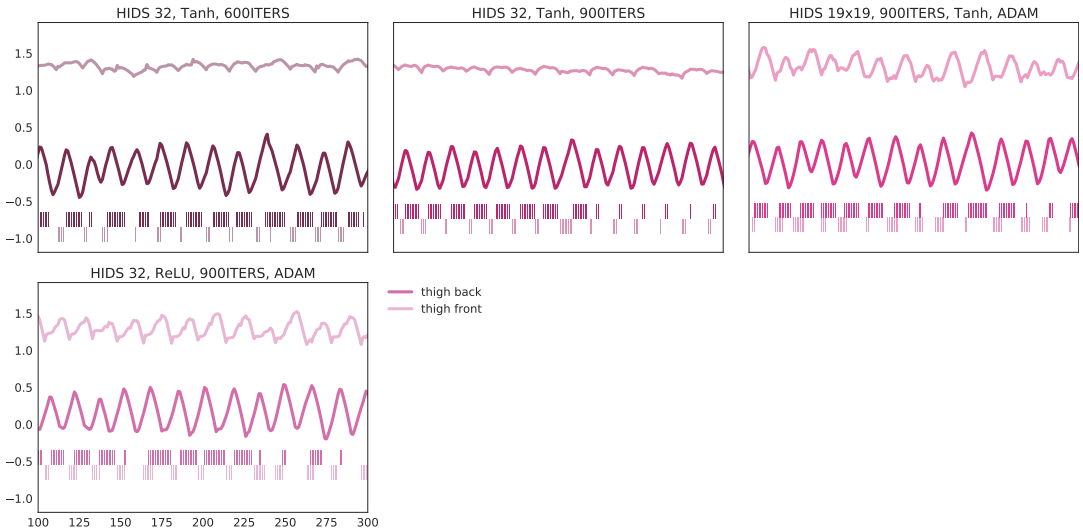


FIGURE 5.8: Evaluation results for 4 different ES models.

Table of results:

MODEL				ROBUSTNESS	STABILITY			PHASE SHIFT
HID	ACT	OPT	ITER		VERT.	PITCH.	VEL.	
32	TANH	SGD	600	6.28	1.15 ± 0.021	0.95 ± 0.0095	$\mathbf{1.20 \pm 0.017}$	4.0 ± 0.010
32	TANH	SGD	900	5.58	1.14 ± 0.019	$\mathbf{0.88 \pm 0.011}$	1.30 ± 0.017	4.4 ± 0.0054
19X19	TANH	ADAM	900	7.04	$\mathbf{1.10 \pm 0.018}$	1.23 ± 0.012	1.28 ± 0.015	3.8 ± 0.0090
32	RELU	ADAM	900	**7.93**	1.32 ± 0.014	1.20 ± 0.013	1.22 ± 0.011	$\mathbf{3.3 \pm 0.0057}$

TABLE 5.2: Evaluation results for 4 different ES models.

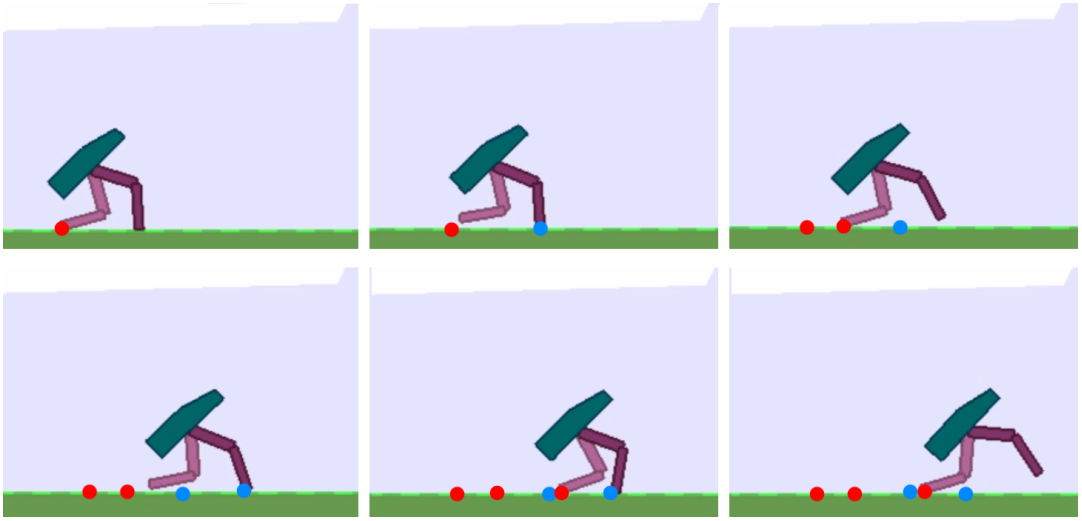


FIGURE 5.9: Time-lapse of the best ES model. Note how the phase of the legs are unnatural because the legs do not overlap.

5.2 Discussion

For a graphical comparison of the best model performance between ES and DDPG with the three evaluation metrics, see Figures 7.1, 7.2, 7.3 in Chapter 7.

The results show an interesting delineation between DDPG and ES. Although both methods achieved maximum *mean return*, based on our walking evaluation metrics, DDPG produced a gait more true to human bipedal walking in comparison to ES, whereas ES produced a gait that was more robust to more difficult terrain. Paradoxically, ES also had the model with the highest-performing return. Why can ES perform so well with a policy that does not walk like a human? From observing the best gaits learned by ES and the best gaits learned by DDPG, it seemed that ES gaits lacked richness in their 'responsiveness'. That is, ES gaits seemed to be performing actions independent of drastic changes in the state space, whereas DDPG appeared to respond more state changes such as sudden falling. From this, we hypothesised that ES learned minimum action sequences that maximised reward without associating the actions with the state input. On the other hand, DDPG learned action sequences which were highly correlated to the state space. To test this hypothesis we plotted the change in action sequences as each element of the state space vector was perturbed, for both ES (Figure 5.10) and DDPG (Figure 5.11).

From the plots, it is clear that DDPG has a more diverse set of action responses relative to the state input vector. We can see that DDPG's optimisation step is dependent on fewer actions:

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}[\nabla_\theta log(\pi(\mathbf{s}))Q(\mathbf{s}_i, \mathbf{a}_i)] \\ &= \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta log(\pi(\mathbf{s}_i))Q(\mathbf{s}_i, \mathbf{a}_i)\end{aligned}$$

Where during training, N is the size of the batch sampled from the buffer. We used $N = 64$, and therefore the parameter update was dependent on 64 actions. On the

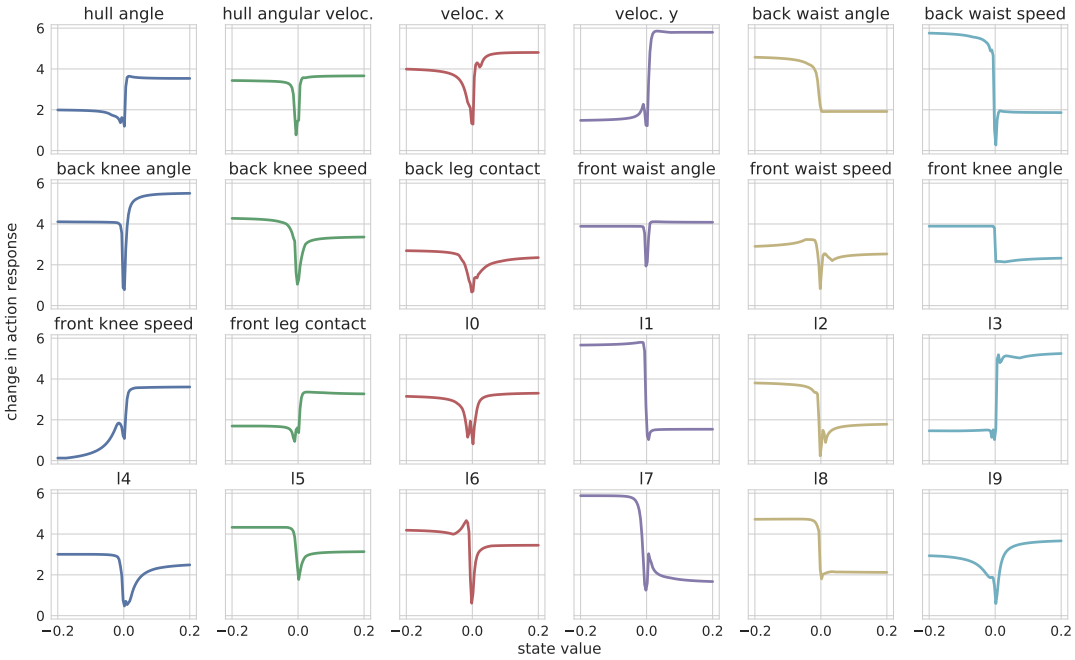


FIGURE 5.10: 'Responsiveness' of the ES model. From an initial zero-vector base state, we perturb each element of the state vector between $[-0.2, 0.2]$ and plot the absolute change in the action output. Beyond $[-0.2, 0.2]$ the actions just saturate, hence why the range is less than the DDPG data.

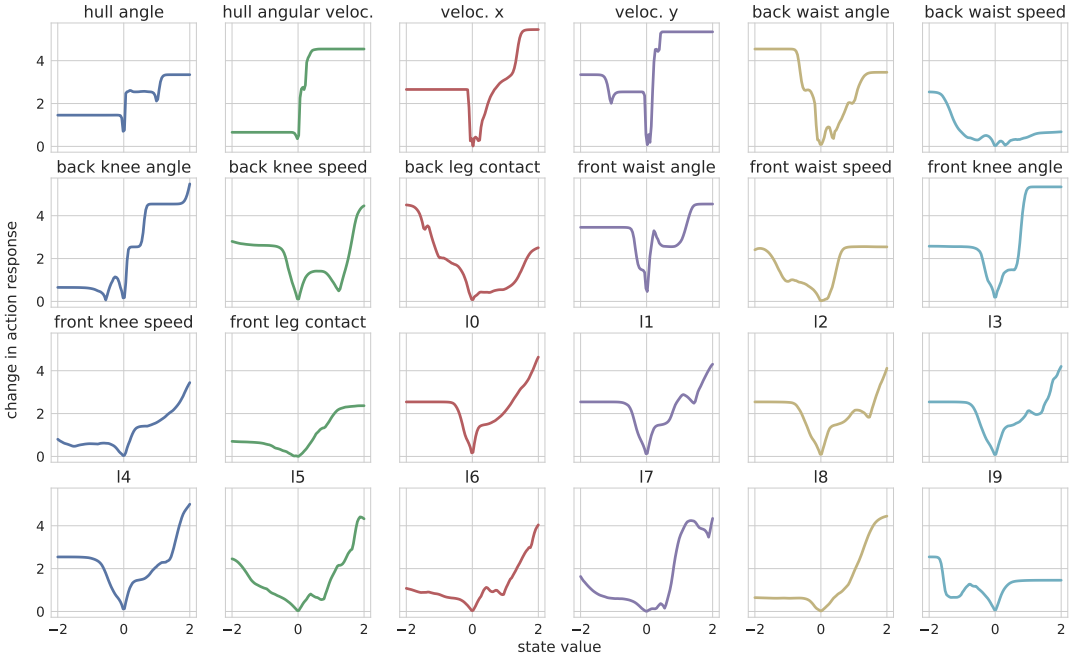


FIGURE 5.11: 'Responsiveness' of the DDPG model. From an initial zero-vector base state, we perturb each element of the state vector between $[-2.0, 2.0]$ and plot the absolute change in the action output. It is clear DDPG has more 'richness' in its responses to change in the state input.

other hand, ES's optimisation step is dependent on more actions:

$$\begin{aligned}\nabla J(\theta) &= \mathbb{E}_{\epsilon\sim\mathbb{N}(0,\mathbb{I})\sigma}[\epsilon \cdot F(\theta+\epsilon)] \\ &= \frac{1}{n}\sum_{i=1}^{n} \epsilon_i \cdot F(\theta+\epsilon_i)] \\ &= \frac{1}{n}\sum_{i=1}^{n} \epsilon_i \cdot G(\{\pi_{\theta+\epsilon_i}(\mathbf{s}_0), ..., \pi_{\theta+\epsilon_i}(\mathbf{s}_T)\})]\end{aligned}$$

Where $G(\{\pi_{\theta+\epsilon_i}(\mathbf{s}_0), ..., \pi_{\theta+\epsilon_i}(\mathbf{s}_T)\})$ is a transformation of $F(\cdot)$ into some return on the total actions under some policy function parameterised by $\pi_{\theta+\epsilon_i}$. From this transformation we can now see the total actions for one optimisation step is $n \cdot T$. Therefore, because the maximum number of timesteps is capped at $\approx$ 1000, for a population size of just 20, ES is dependent on $\approx 20,000$ actions in the worst case. The uniqueness of each action is fundamentally smoothed because the sample size of actions is so large, making it a challenge for ES to learn specialised actions from associating parameters and states with action responses. The return, which is the sum of the rewards over time $G_t = \sum_{i=1}^{\inf} \gamma^{i-1} R_{t+i}$, eliminates information about how single actions contribute to the success of a population individual. This analysis provides a potential reason why DDPG has a policy function tuned towards very specific action behaviour for different states, while ES learns more general actions. Unlike ES, DDPG is optimised relative to a small batch of actions, where a return is associated with each action.

Chapter 6

Towards a More Natural Bipedal Gait

The results in Chapter 5 showed that although various models conceivably obtain maximum *mean return* in an environment, the reward function alone does not necessarily produce the desired behaviour of a human-like walking gait. If we want to look beyond perfectly-tuned reward functions which do not generalise well [72], we need to explore ways in which either the *model* or the *algorithm* can be adapted to encourage more complex emergent behaviour. Some interesting research in this area has included trying to remove the reward function entirely and alternatively maximising the entropy of the model [85]. Other research has included changing the models by designing neural architectures similar to those found in animals and humans [19] [88] [21], or using hierarchical models [50] [78] [16]. We focused on the second example, changing the model architecture, and use the same algorithms, DDPG and ES, and the same reward function.

6.1 Enhanced Model Architectures

In this section we outline our exploration of two concepts to potentially help produce a more natural gait:

1. The ES algorithm is versatile because it doesn't compute exact gradients. We can therefore use functions that are non-differentiable and non-continuous *or* are even too sensitive to compute stable gradients. Recurrent Neural Networks (RNNs) are infamous for their difficult gradients when adjusting the weights via backpropagation through time (BPTT). LSTMs were created to tame the exploding/vanishing gradient problem that results from BPTT [27]. We evaluated the affects of incorporating recurrent architectures into the ES model. We experimented with some interesting architectures including *Hopfield Networks*, *Binary Networks*, and *Echo State Networks*, but we only found time to rigorously explore two recurrent models: *Fully-Recurrent Networks* and *Jordan networks* [33] [32].

2. The action-output of the neural network model was not biologically-plausible for the task we were trying to accomplish. Research into the nervous system of walking mammals shows that systems such as central pattern generators (CPGs) and distributed, independent muscle control exist (see Section 2.3.2 in Chapter 2). These sub-systems work in isolation of immediate control from the brain. Taking inspiration from biology, we introduced different novel control

constraints on the model to see how they affected the gait that was learned. We used DDPG to train these models, as it produced better walking policies than ES. Specifically, we explored a two-output symmetrical model, a hierarchical-controller model, and a distributed-controller model.

6.2 Exploring Recurrent Architectures using Evolutionary Strategies

Some work has shown that ES struggles to deal with recurrent architectures [18]. But, due to the clear advantage that ES could provide for recurrent architectures by not having to compute gradients, we were still interested in exploring how it would fare. We explore two types of recurrent architectures: a *Fully-Recurrent Network* and a *Jordan Network*.

6.2.1 Fully-Recurrent Network Model

The Fully-Recurrent (FR) model incorporates a simple set of recurrent weights into the hidden layer which store the state of the previous input and feed back into the hidden units at the next input. Unlike a feed-forward neural network, this enables the model to have 'memory' and utilise past information.

6.2.2 Jordan Network Model

The Jordan Network (JN) maintains an external memory 'context' layer that receives input from the output of the network. This memory state recurrently feeds back into itself and into the hidden layer of the next input. The input-weights into the context layer are all 1, but the output weights can be adjusted. We chose the Jordan Network because of its interesting architecture and it has also been used for evolutionary algorithms before [71].

6.2.3 Results

Successfully training the recurrent models proved a challenge. The models suffered from longer training times, due to having more parameters to optimise, and exhibited inconsistent results for similar hyperparameters. We found that decreasing the noise from $\sigma = 0.1$ to $\sigma = 0.01$ and using larger populations helped. We present the five best models between both the Jordan Network and the Fully-Recurrent Network. Disappointingly, the best recurrent policies were qualitatively similar to the unnatural feed-forward policies learned by the ES algorithm. Most interestingly, the ES algorithm could, in theory, scale all the recurrent weights to values near zero and learn the original, better-performing feed-forward model, but instead all models had uniformly-scaled weights.

Results of robustness: The robustness of the recurrent models was poor especially in comparison to the robustness of the feed-forward ES models. We hypothesise that the recurrent architecture only hindered the 'richness' of the learned policy.

From our evaluation of the feed-forward ES model in Chapter 4, we found that actions were highly independent from the state input. On reflecting on this, adding a recurrent layer could *further* enforce this independence on the current state input.

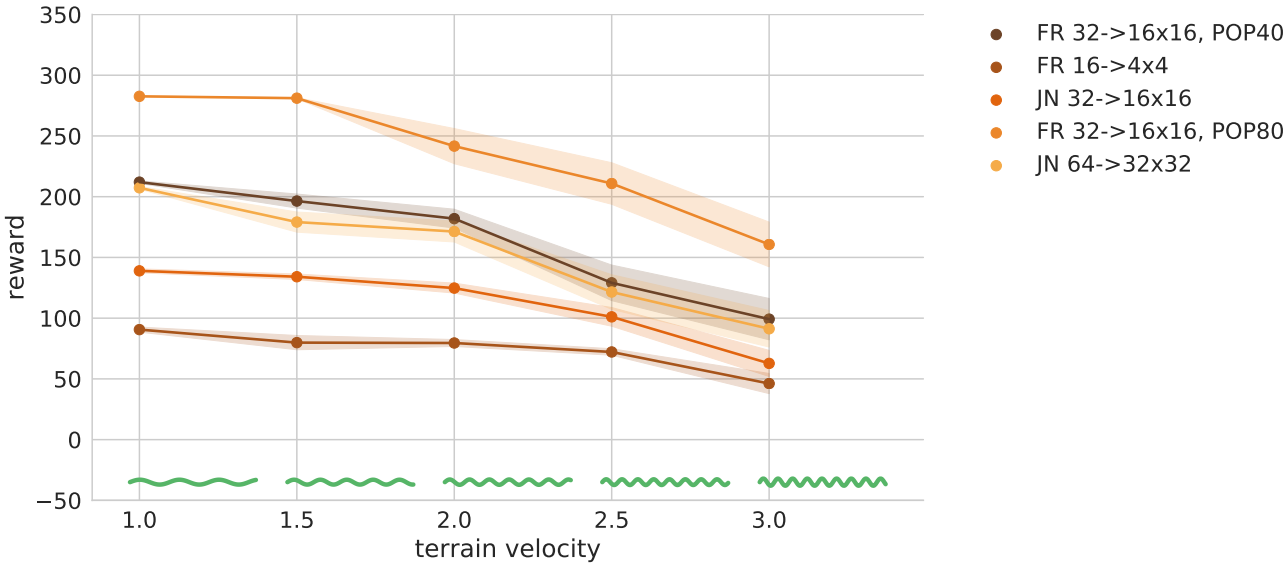


FIGURE 6.1: Robustness results for 5 different RNN models.

Results of steady-state stability: The steady-state stability of the recurrent models was the best metric. This is because the walking policy learned by the model displayed clear cyclic behaviour independent of the state input. As this cyclic behaviour was already common from the feed-forward model, the recurrent architecture would only enforce this cyclic behaviour. This result shows that recurrent models can produce stable oscillations in the movement - even if it is not a walking gait.

FIGURE 6.2: Steady-state stability results for 5 different ES-RNN models.

Results of phase-shift symmetry: The phase-shift symmetry portrays the failure of the recurrent model well. The recurrent models all learned very similar policies

which involved a 'crawling' cyclic movement analogous to the time lapse of the feed-forward ES model in Figure 5.9.

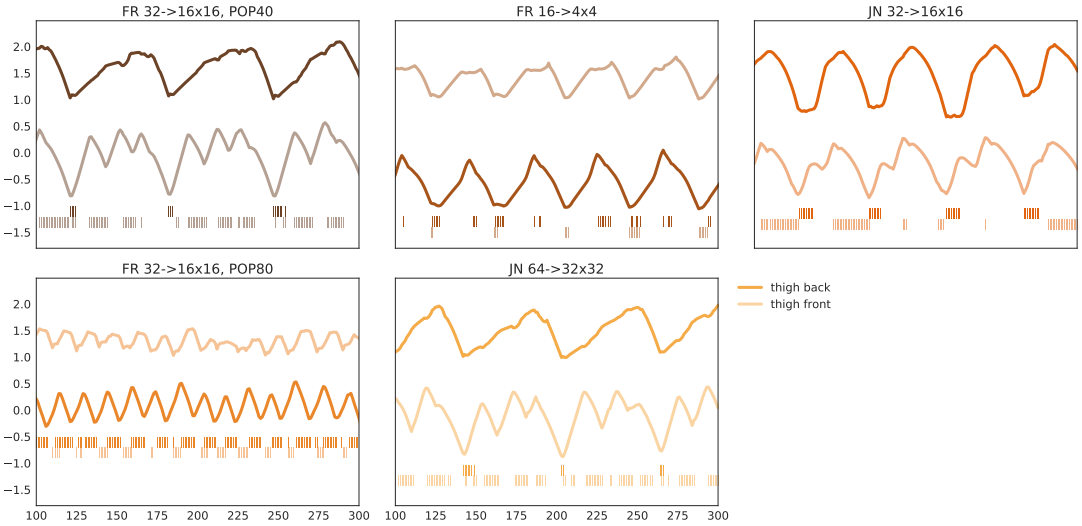


FIGURE 6.3: Phase-shift symmetry results for 5 different ES-RNN models.

Table of results:

MODEL			ROBUSTNESS	STABILITY			PHASE SHIFT
RNN	HIDS	POP		VERT.	PITCH.	VEL.	
FR	32->16x16	40	4.97	1.18 ± 0.019	1.37 ± 0.017	1.26 ± 0.029	3.13 ± 0.0093
FR	16->4x4	80	2.29	1.28 ± 0.012	1.25 ± 0.013	1.20 ± 0.033	3.19 ± 0.0023
JN	32->16x16	80	3.44	1.48 ± 0.018	1.46 ± 0.013	1.46 ± 0.027	3.39 ± 0.0052
FR	32->16x16	80	7.32	0.83 ± 0.014	1.13 ± 0.014	0.98 ± 0.020	3.54 ± 0.019
JN	64->32x32	80	4.65	1.19 ± 0.019	1.41 ± 0.014	1.24 ± 0.021	3.19 ± 0.0078

TABLE 6.1: Results for the five ES-RNN models.

6.3 Exploring Multi-Controller Architectures using DDPG

We developed three unique models, inspired by the biology of a real-life human gait. The first model, the *Two Output Symmetrical (TOS)* model, was a simple model with only action outputs which controlled only a single leg; the other leg was forced to move symmetrically. The second model combined the TOS model and the original 4-action output model to create an end-to-end trainable *Hierarchical Controller (HC)* model. The 4-action output model was a high-level controller and the TOS model was a low-level controller. We then generalised the HC model to have a 'rotating' hierarchy during training, across six different controllers. This *Distributed Controller (DC)* model constrained each controller to a subset of the original action outputs.

6.3.1 Two-Output Symmetrical Model

The Two-Output Symmetrical (TOS) model is an overly-simplified heuristical model that exploits the symmetry of legs during a bipedal walk. The model controls

only a single leg via two action outputs. The two outputs correspond to the torques of one waist and one knee joint, with the other leg acting only by inverting the sign of the torque values from the controlled leg.

Formally, the model is defined by a neural network with an input of the state vector $s \in \mathbb{R}^{24}$ and an output of $\boldsymbol{\tau} \in \mathbb{R}^2$. The action vector is then defined as $\boldsymbol{a} = (\tau_0, \tau_1, -\tau_0, -\tau_1)$. Hence the symmetry of the torques of the waist joints is forced to be symmetrical because $\tau_0 - \tau_0 = 0$. Although the knee joints do not have to be symmetrical, by constraining them to be it allows us to further simplify the model. The reward function is still penalised for the entire action of all four joints.

Results

Results of robustness: The TOS model slightly improved the robustness of the 4-output model, already showing that the 4-output model was not good. Despite the 4-output model having symmetrical walking, it was a robust design. The TOS model is robust because the adaption to new terrain does not require learning individual dynamics for each joint. Instead, the actions can be adjusted for a single leg and will immediately transfer to the alternate leg, allowing for some scalability of the gait on new terrain.

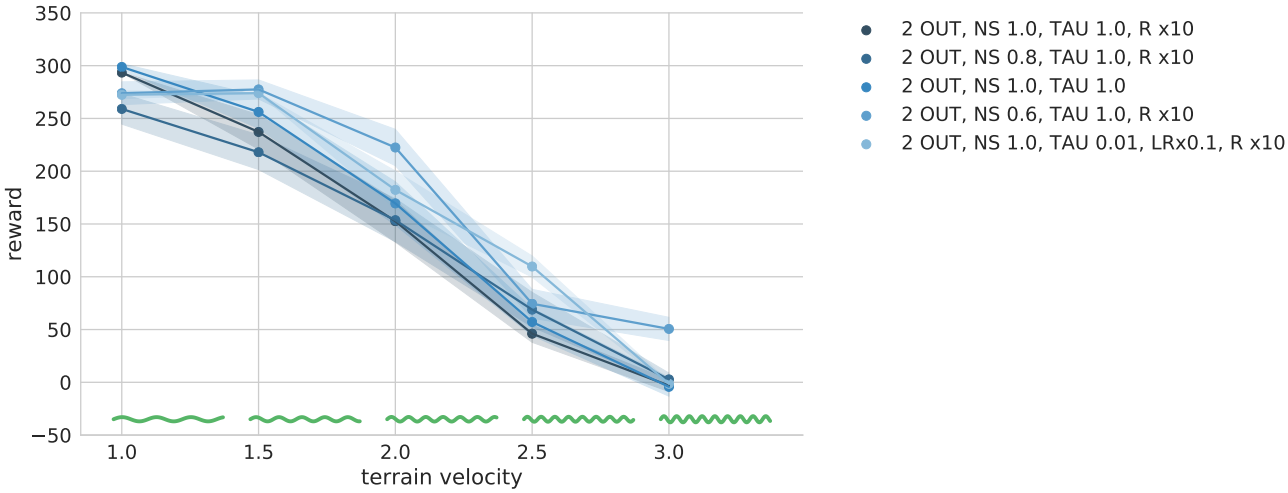


FIGURE 6.4: Robustness results for 5 different TOS models

Results of steady-state stability: Naturally, the TOS model has an unstable gait. Because the model is over-constrained, it cannot appropriately adjust its knee joints separately to provide stability. This causes the head to rock back and forth and the forward movement to be abrupt.

Results of phase-shift symmetry: Because the TOS model is constrained to have symmetrical control, one would think it would perform well on the phase-shift symmetry evaluation. The best model performed on par with the 4-output model, but the other models did not perform as well. This is largely due to the fact that the model did not learn how to cross-over its legs in the 2D environment, and hence learned a sub-optimal 'shuffling' gait for some hyperparameter settings. On further reflection, this could have been due to the reward function penalising the movement

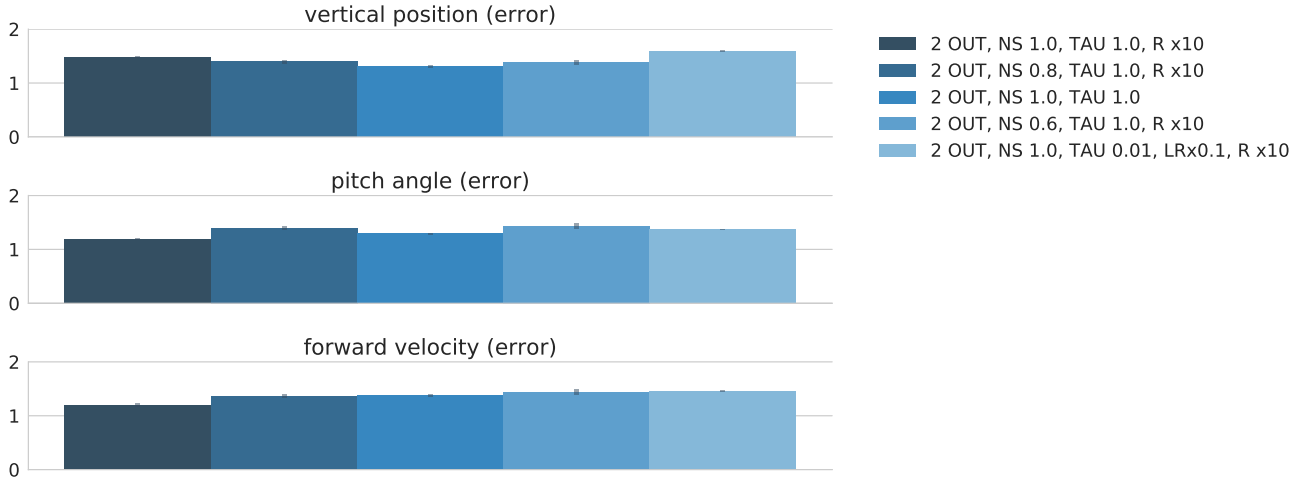


FIGURE 6.5: Steady-state stability results for 5 different TOS models

of all legs which resulted in twice the penalty for a single output:

$$R(t) = \alpha(x(t) - x(t-1)) - \beta \sum_{i=0}^{3} |a_i| - 100I\{y(t) < p_y\} \tag{6.1}$$

$$R(t) = \alpha(x(t) - x(t-1)) - \beta(2|\tau_0| + 2|\tau_1|) - 100I\{y(t) < p_y\} \tag{6.2}$$

For future experiments we would recommend scaling the penalty proportionally.

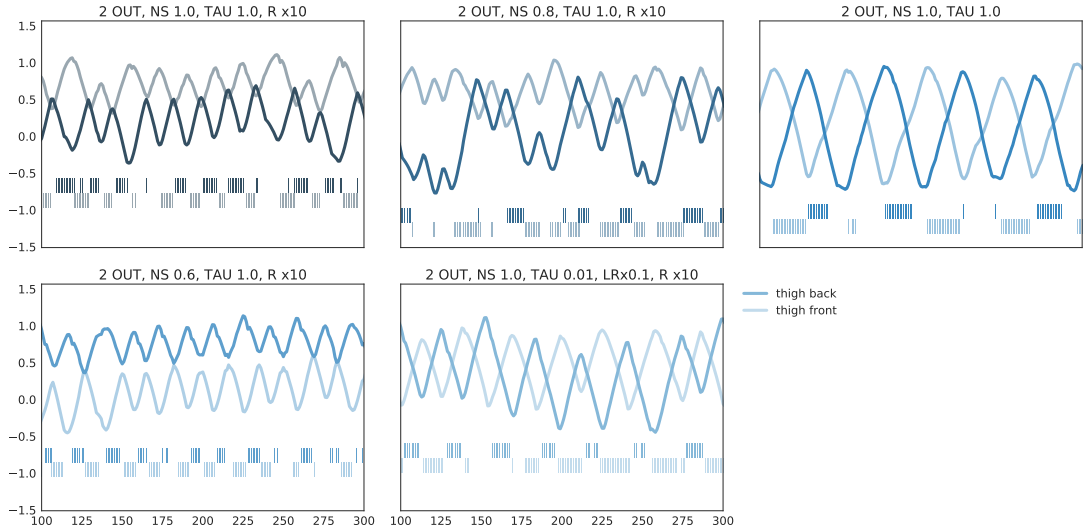


FIGURE 6.6: Phase-shift symmetry results for 5 different TOS models

Table of results:

MODEL				ROBUSTNESS	STABILITY			PHASE SHIFT
NS	TAU	RS	LR (CR/AC)		VERT.	PITCH.	VEL.	
1.0	1.0	10	0.001/0.0001	3.53	1.47 ± 0.017	1.88 ± 0.016	$\mathbf{1.20 \pm 0.029}$	1.69 ± 0.014
0.8	1.0	10	0.001/0.0001	3.58	1.40 ± 0.032	1.39 ± 0.031	1.36 ± 0.033	1.83 ± 0.023
1.0	1.0	1	0.001/0.0001	3.84	$\mathbf{1.31 \pm 0.025}$	$\mathbf{1.29 \pm 0.016}$	1.38 ± 0.027	0.90 ± 0.022
0.6	1.0	10	0.001/0.0001	**4.91**	1.38 ± 0.041	1.43 ± 0.047	1.44 ± 0.047	2.14 ± 0.016
1.0	0.01	10	0.01/0.001	4.39	1.59 ± 0.015	1.37 ± 0.0089	1.46 ± 0.012	$\mathbf{0.78 \pm 0.0096}$

TABLE 6.2: Results for five TOS models.

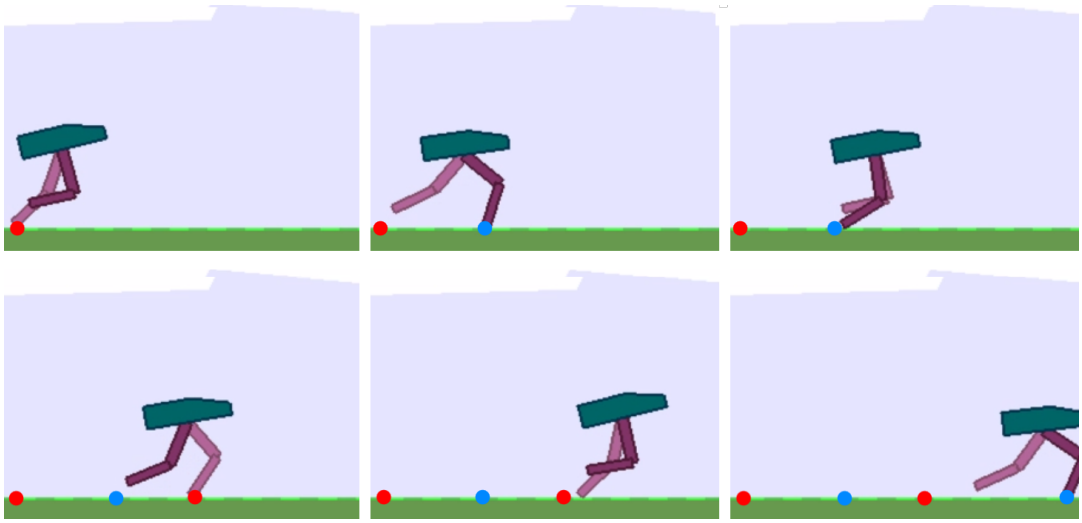


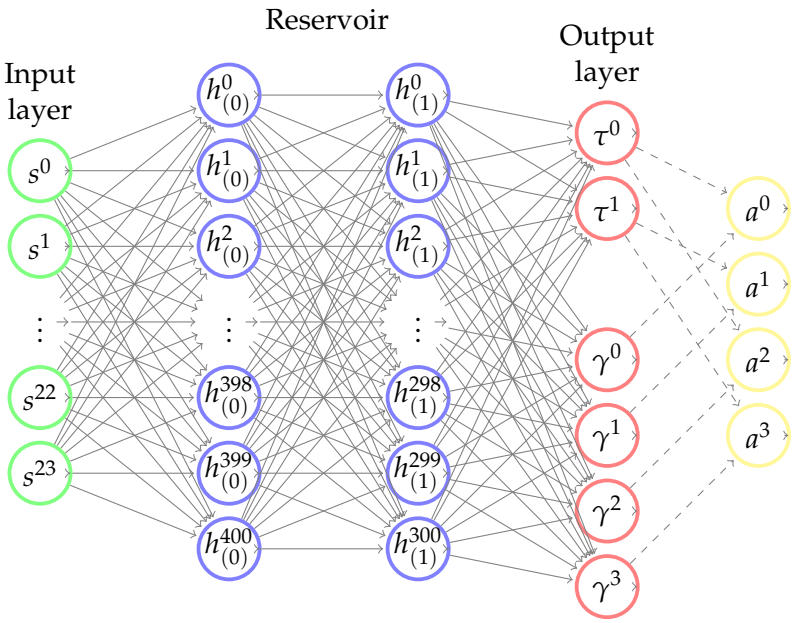
FIGURE 6.7: Time-lapse of the best TOS model.

6.3.2 Hierarchical-Controller Model

The Hierarchical-Controller (HC) model is inspired by the disconnect between low-level cyclic behaviour in the legs and the high-level control system in the brain. In a hierarchical fashion, one controller is the simple 'cyclic' two-output model described earlier, and the second controller is an 'error-correcting' controller which adjusts the legs when the cyclic behaviour needs to be broken - for example if the agent is falling and needs to catch itself. Biologically, the model is related to the relationship of CPGs (as discussed in Section 2.3.2 of Chapter 2) to that of the brain. The low-level controller acts as a 'CPG' by enforcing cyclic control on the legs, with some feedback from the environment. The high-level controller is an error-correction signal from the 'brain', which adjusts the gait accordingly. We use shared hidden-layers for the model to speed-up training time [38] and the output of the network is a vector for each controller: $\boldsymbol{\tau} \in \mathbb{R}^2$ and $\boldsymbol{\gamma} \in \mathbb{R}^4$. The final action vector is therefore: $\boldsymbol{a} = (\tau_0 + \gamma_0, \tau_1 + \gamma_1, -\tau_0 + \gamma_2, -\tau_1 + \gamma_3)$. A diagram of the HC model is shown below. At first inspection, this model is no different from the 4-action model we used earlier. The CPG network can learn to output zero-values and the error-correction network will solve the problem independently. We avoid this behavior through penalising the use of the error-correction network by disproportionately scaling the output of the network in the reward function:

$$R(t) = \alpha(x(t) - x(t-1)) - \beta_0 \sum_{i=0}^{2} |\tau_i| - \beta_1 \sum_{i=0}^{4} |\gamma_i| - 100I\{y(t) < p_y\}$$

We constrained the values for β_1 such that: $\beta_1 = \beta_0 + \beta_0 \times 10^k$. The best results were when $k = 2$. Note that $\beta_0 = 0.028$ was still the same from the original reward function.



During initial experiment with the HC model, we found it was unable to train consistently due to the separate controllers being unable to learn robust independent policies. We found that turning the low-level controller *off*, intermittently, improved convergence drastically. We experimented with training either *both controllers* together longer or training the low-level controller by itself longer. When labeling the model, we use the notation $1 : N$ to mean "every N training updates, do an update with both controllers, otherwise train with just the low-level controller." $N : 1$ means "every N updates, do an update with just the low-level controller, otherwise train with both controllers."

Results

We first show that the behaviour of the model matches our theory behind the model. In Figure 6.8, we see a time slice of the model being run on a terrain with an acceleration of 2.5. There are clear spikes on the high-level controller actions where adjustments are made to the joints - these directly coincide with stabilising the legs when the agent is going up or down hills.

Results of robustness: Incredibly, the HC model produced drastic improvement in the robustness score. The best models were scoring > 250 reward on an unseen terrain of velocity 2.5. The best model, trained with $7 : 1$, is almost a combination of the best TOS model and the best 4-output model, resulting in high robustness and high stability.

Results of steady-state stability: For some of the HC models, the stability is as poor as the TOS model, but the best TOS model had stability as low as the 4-output model.

Results of phase-shift symmetry: The $7 : 1$ HC model was also competitive with the best 4-output model for phase-shift symmetry analysis. Most likely this model found a good balance between engaging the low-level controller for symmetry, while falling back to the high-level controller for robustness on high-velocity terrains.

Table of results:

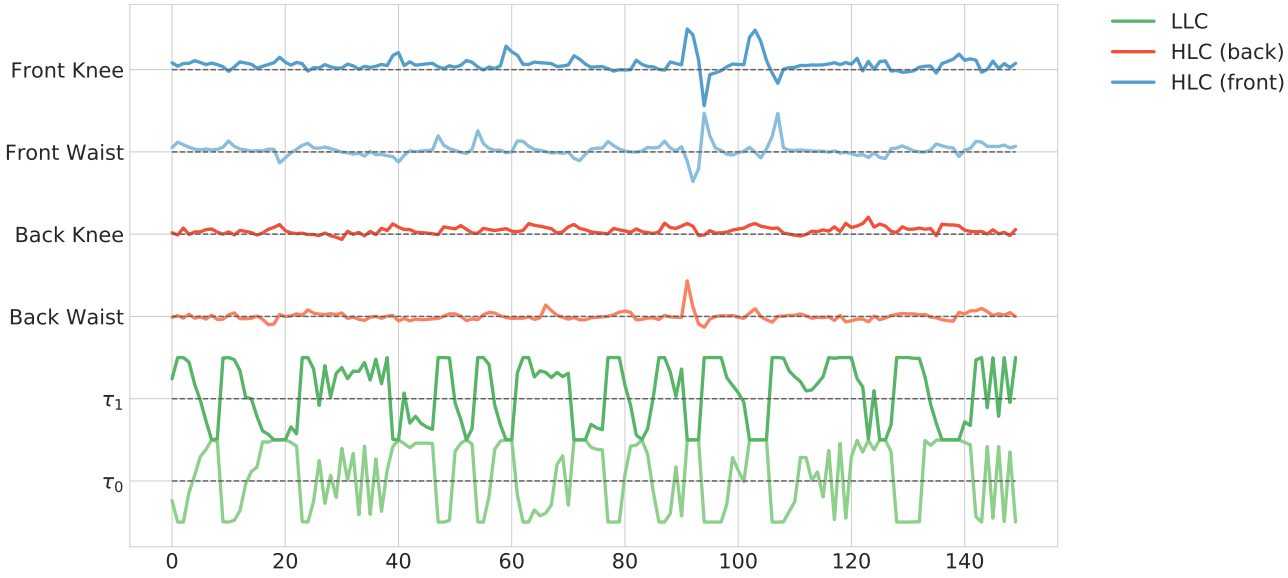


FIGURE 6.8: Sample actions for the best HC model.

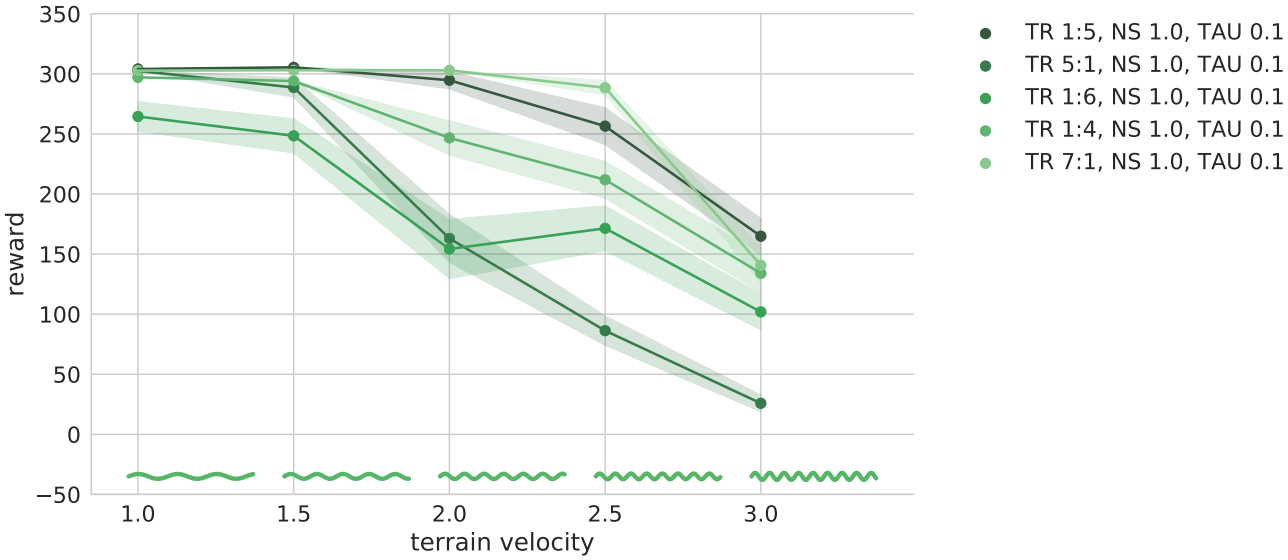


FIGURE 6.9: Robustness results for 5 different HC models.

MODEL		ROBUSTNESS	STABILITY			PHASE SHIFT
TAU	TR		VERT.	PITCH.	VEL.	
0.1	1:5	8.30	1.43 ± 0.018	1.39 ± 0.014	1.37 ± 0.014	1.52 ± 0.0084
0.1	5:1	4.52	1.30 ± 0.054	1.30 ± 0.050	1.32 ± 0.034	1.48 ± 0.025
0.1	1:6	5.60	1.40 ± 0.034	1.34 ± 0.030	1.38 ± 0.028	1.81 ± 0.015
0.1	1:4	7.21	1.31 ± 0.032	1.26 ± 0.022	$\mathbf{1.04 \pm 0.025}$	1.24 ± 0.030
0.1	7:1	**8.35**	$\mathbf{1.27 \pm 0.023}$	$\mathbf{1.24 \pm 0.012}$	1.20 ± 0.022	$\mathbf{0.47 \pm 0.0055}$

TABLE 6.3: Results for five HC models.

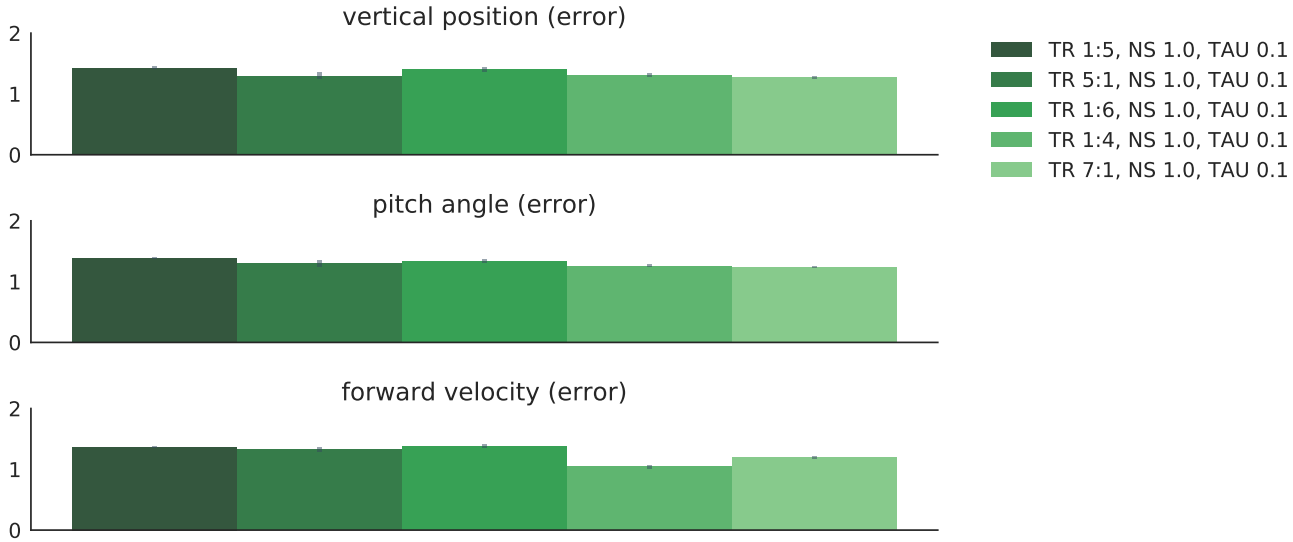


FIGURE 6.10: Steady-state stability results for 5 different HC models.

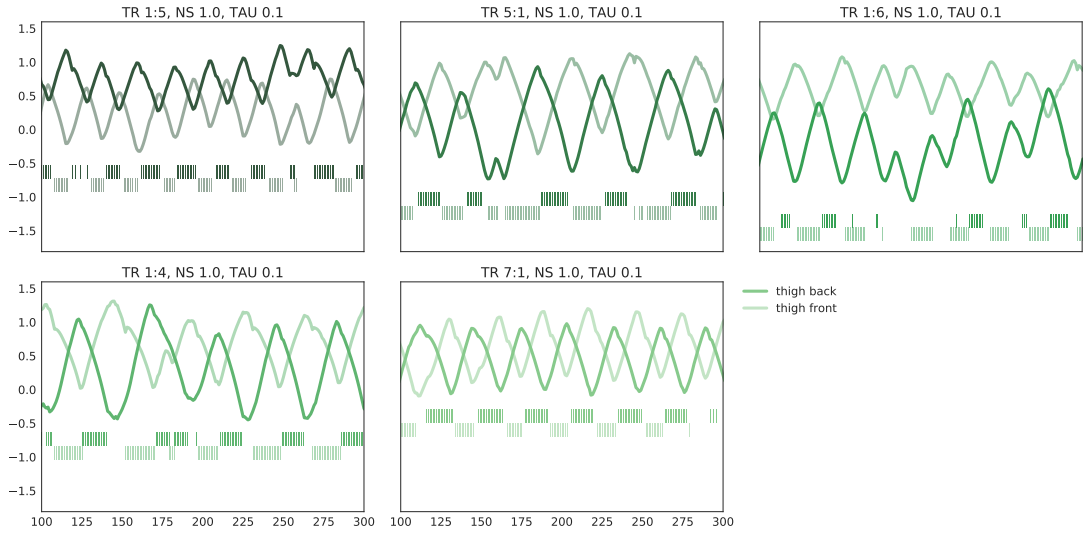
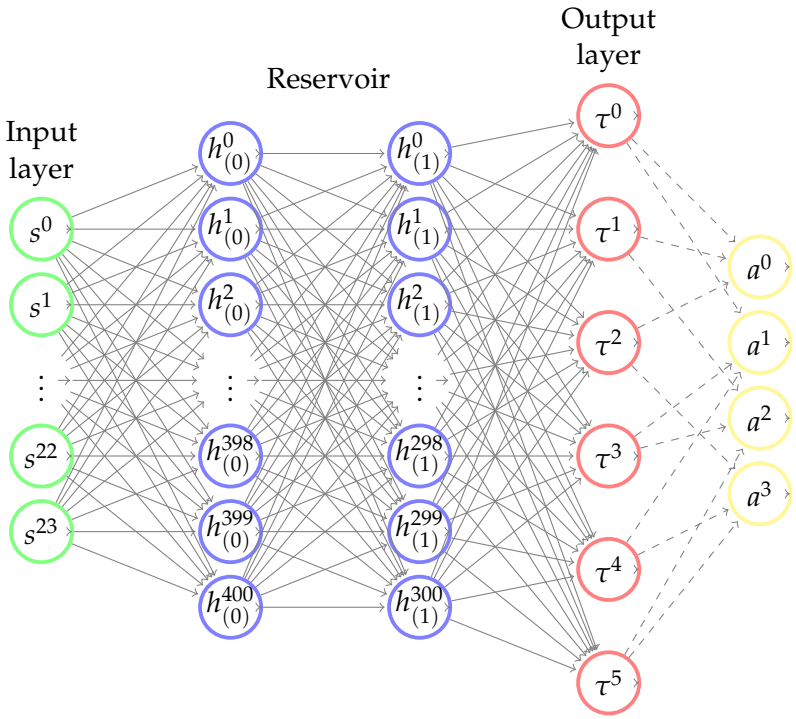


FIGURE 6.11: Phase-shift symmetry results for 5 different HC models.

6.3.3 Distributed-Controller Model

The Distributed-Controller (DC) generalises the Hierarchical-Controller model and uses competition between controllers as well as reward penalties to learn cooperative behaviours. The DC is not a hierarchy - alternatively, each controller contributes to only two actions, with one controller for each combination of two actions. Unlike the HC model, controllers are constantly switching places in the hierarchy. We took inspiration from *D. Bucher et al*'s work researching modular rhythmic control in the body [11]. The DC model is depicted below.



Similar to the HC model, training the DC model required a unique process. Because there is no hierarchy, we cannot prevent the model from just learning the 4-output model behaviour, even with the reward adjustment used to train the HC model. The solution was to introduce a 'mask' of uniform binary values, $\boldsymbol{m} \in \mathbb{R}^{|\mathbf{a}|}$, where $m_i = I\{U(0,1) < \rho\}$, $U(0,1)$ is a uniform random value from $(0,1)$ and $\rho = 0.2$. The mask is element-wise multiplied by the actions, such that: $a'_i = a_i \cdot m_i$ for $i = 1,..,4$. The mask is redefined every T optimisation steps, which we set as a parameter of the model. We found $T \approx 30$ worked well. The reward function is also slightly adjusted to penalise the outputs of the controllers instead of the outputs of the actions; this forces the network to minimise its use of controllers that have been 'masked-out':

$$R(t) = \alpha(x(t) - x(t-1)) - \beta \sum_{i=0}^{6} |\tau_i| - 100I\{y(t) < p_y\}$$

Without this training method, the DC model performed worse than the 4-output model.

Results

Plotting the actions of each controller helps give insight into what behaviours the distributed system learned. A time slice of actions while walking on zero-velocity terrain is shown in Figure 6.12.

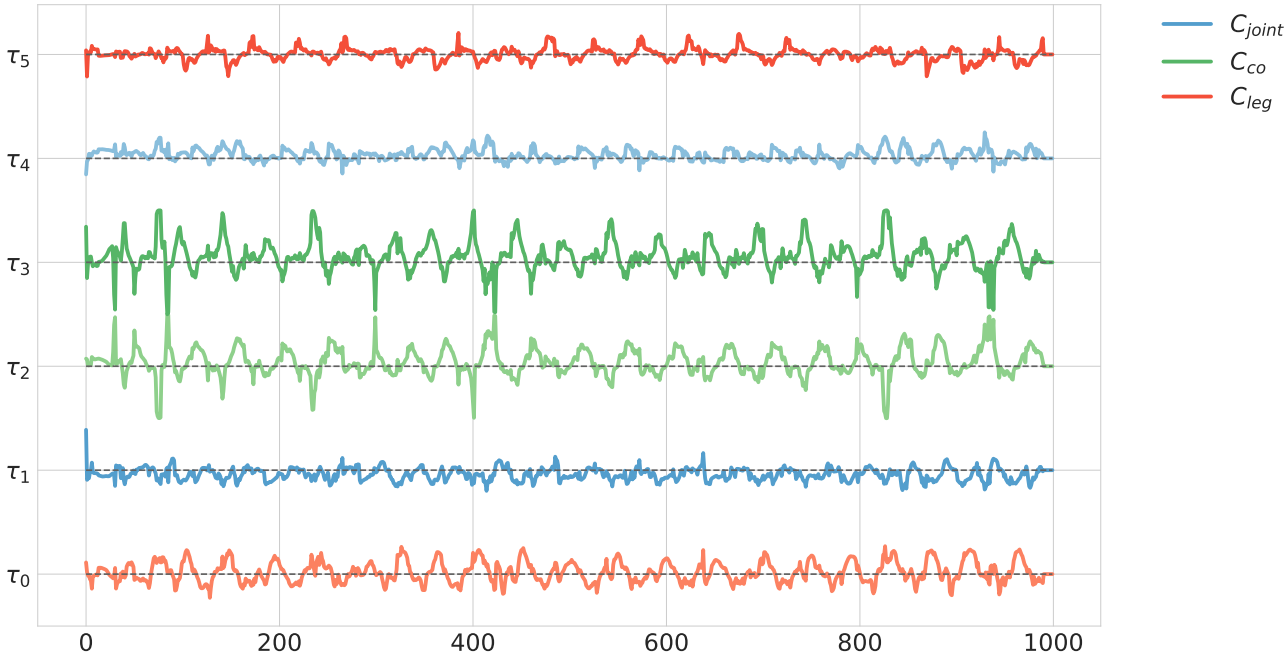


FIGURE 6.12: Sample actions from the best DC model.

From the distributed control system, three distinct classes of controller emerge:

1. *Leg Controller (red):* these controllers had control access only within a single leg, therefore learned relationships between the waist and knee movement.
2. *Joint Controller (blue):* these controllers had control access only on one joint type (either waist or knee), therefore learned relationships between the waist and knee movement.
3. *Coordination Controller (green):* these controllers had control access between opposite joints (back waist $\rightarrow$ front knee, back knee $\rightarrow$ front waist). As the waist is moved forward, we want the opposite waist to move backwards while the opposite knee should move *forwards* so as to keep the support leg as straight as possible. The symmetry of the coordination controllers reflect how well they learned the symmetry of a true bipedal gait.

Results of robustness: The DC controller was the most robust of all the models, while still displaying a natural gait - unlike how ES was robust but had an unnatural gait. We believe this was because the DC controller was an ensemble of low-level controllers such that it was robust to noise in any of the controllers. Because the training process relied on disengaging various controllers, the DC model was able to recover from incorrect actions from other controllers on tough terrain.

Results of steady-state stability: In general, the DC model is more stable than the HC model. Due to the training method for DC, some of the controllers were not in sync, and this would cause anomalous jerks on velocity zero terrain. This caused the

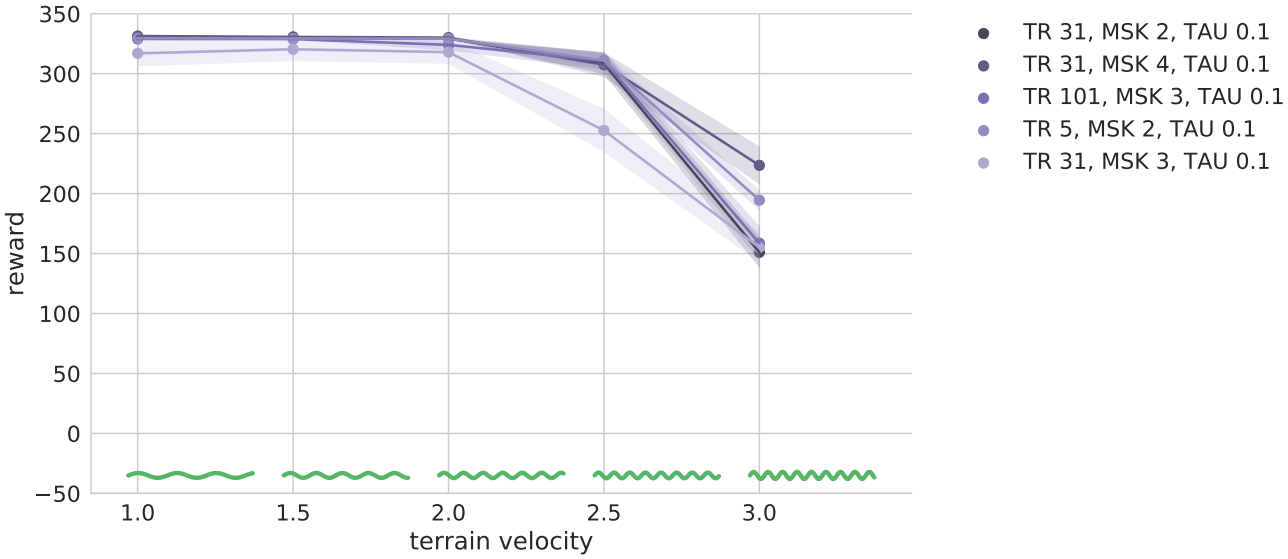


FIGURE 6.13: Robustness results for 5 different DC models

stability error to rise more than is sometimes present on terrain with higher velocity.

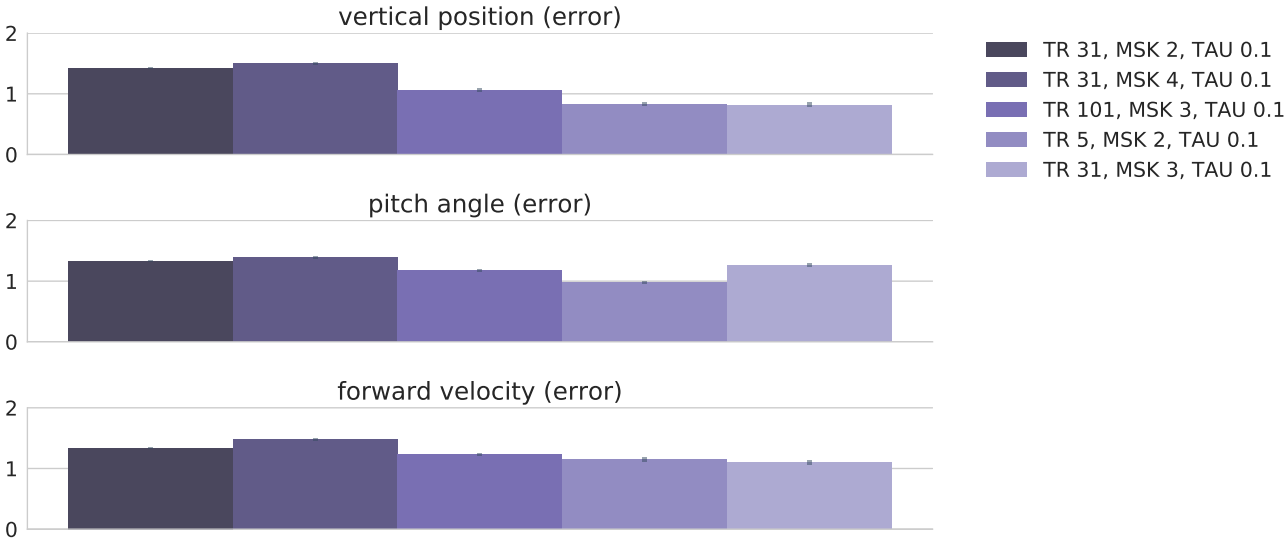


FIGURE 6.14: Steady-state stability results for 5 different DC models

Results of phase-shift symmetry: According to our metric, the phase of the DC model is impressive. The size of the phase is much larger than the original DDPG model at approximately $30 - 40$ time steps versus $15 - 20$ time steps. This made the gait more human-like, as shown in Figure [**fig:dc_timelapse**].

Table of results:

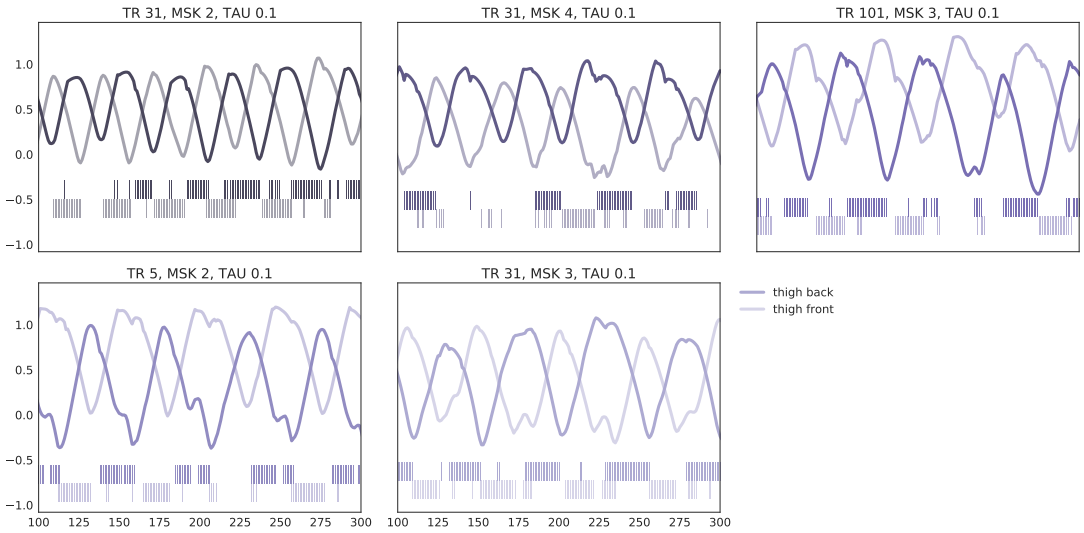


FIGURE 6.15: Phase-shift symmetry results for 5 different DC models

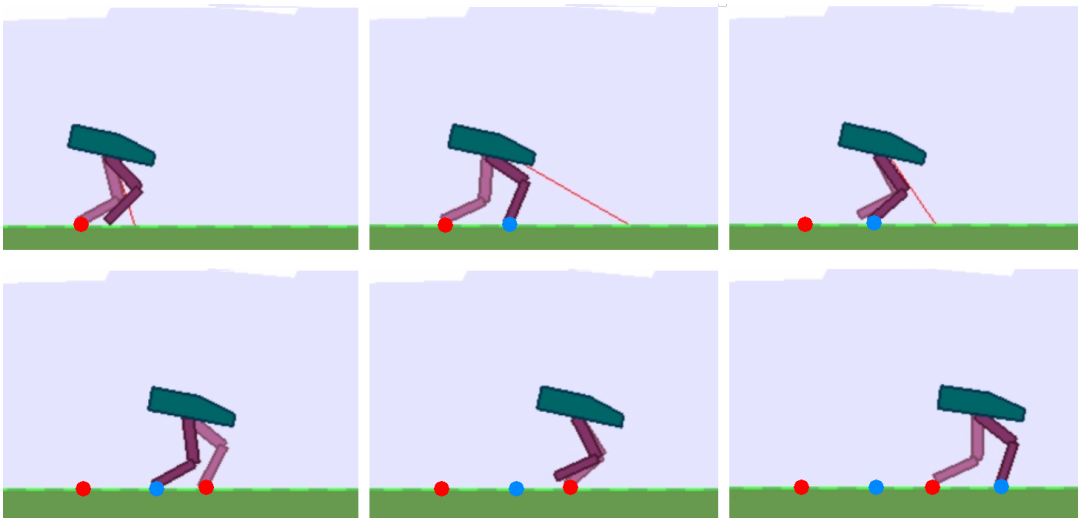


FIGURE 6.16: Time-lapse of the best DC model.

MODEL			ROBUSTNESS	STABILITY			PHASE SHIFT
TAU	TR	MSK		VERT.	PITCH.	VEL.	
0.1	31	2	9.03	1.42 ± 0.015	1.33 ± 0.016	1.34 ± 0.015	$\mathbf{0.42 \pm 0.0055}$
0.1	31	4	**9.76**	1.50 ± 0.022	1.40 ± 0.018	1.48 ± 0.020	1.12 ± 0.010
0.1	101	3	9.08	1.06 ± 0.029	1.18 ± 0.022	1.23 ± 0.018	0.94 ± 0.011
0.1	5	2	9.48	$0.84. \pm 0.026$	$\mathbf{0.97 \pm 0.022}$	1.15 ± 0.035	0.85 ± 0.0047
0.1	31	3	8.44	$\mathbf{0.82 \pm 0.035}$	1.27 ± 0.032	$\mathbf{1.10 \pm 0.035}$	0.67 ± 0.013

TABLE 6.4: Results for five DC models.

Chapter 7

Comparison of Best Models

In this chapter we gather the best models from the original DDPG and ES architectures, as well as the new architectures we explored, and compare their walking policies through our three evaluation metrics. We *briefly* provide some comments and discussion, as most of the conclusive remarks can be found in the following chapter and analysis of each model can be found in its respective chapter.

7.0.1 Comparative Results

We determine the best model from each class by choosing the model in a class that performed best *for each* evaluation metric. For example, some DC models performed very well on the robustness metric, but less well on the phase-shift metric, we therefore would instead prefer a DC model that is slightly worse on robustness but better on phase-shift.

Results of robustness:

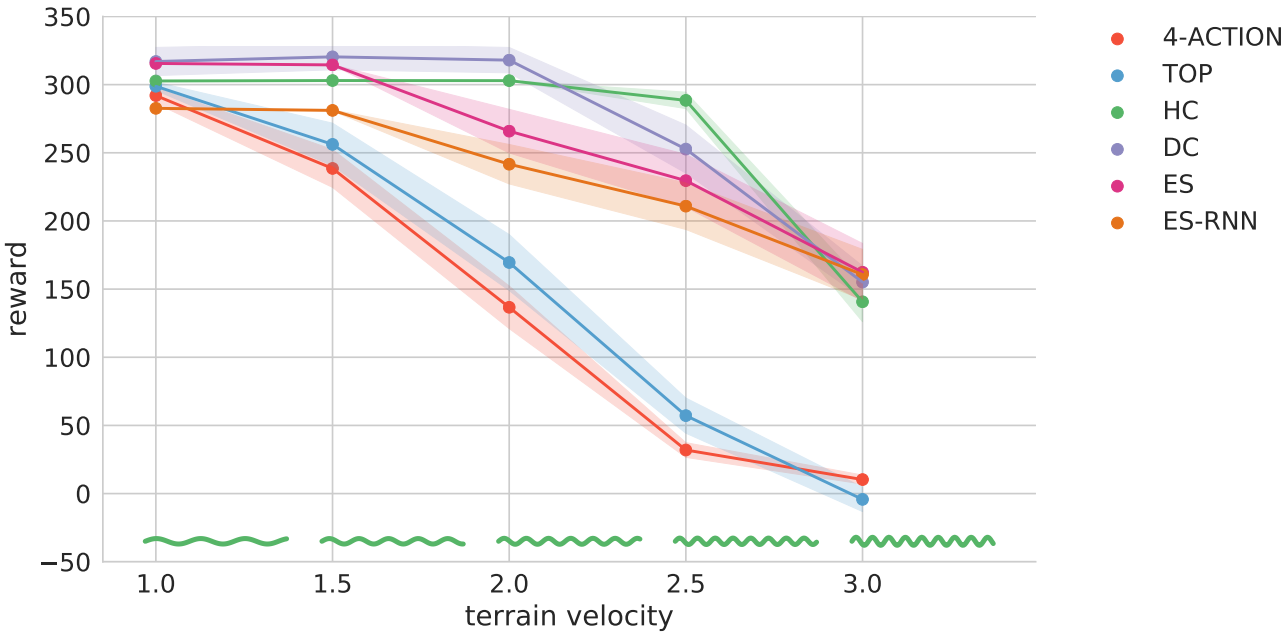


FIGURE 7.1: Robustness results for the best models.

Results of steady-state stability:

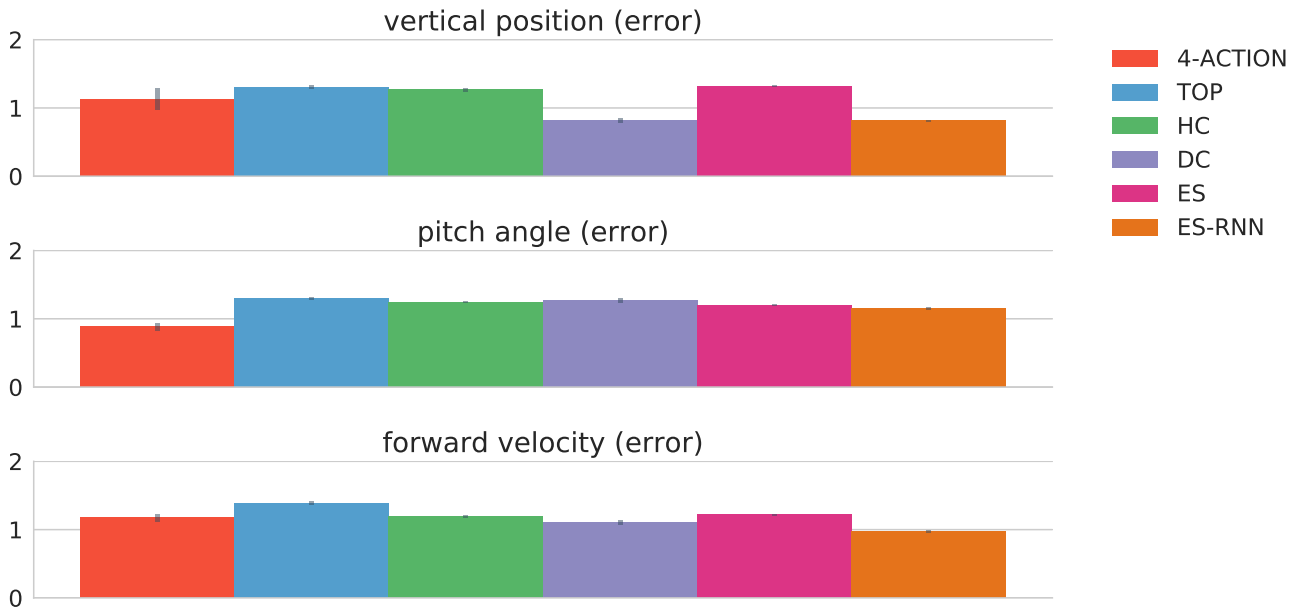


FIGURE 7.2: Steady-state stability results for the best models.

Results of phase-shift symmetry:

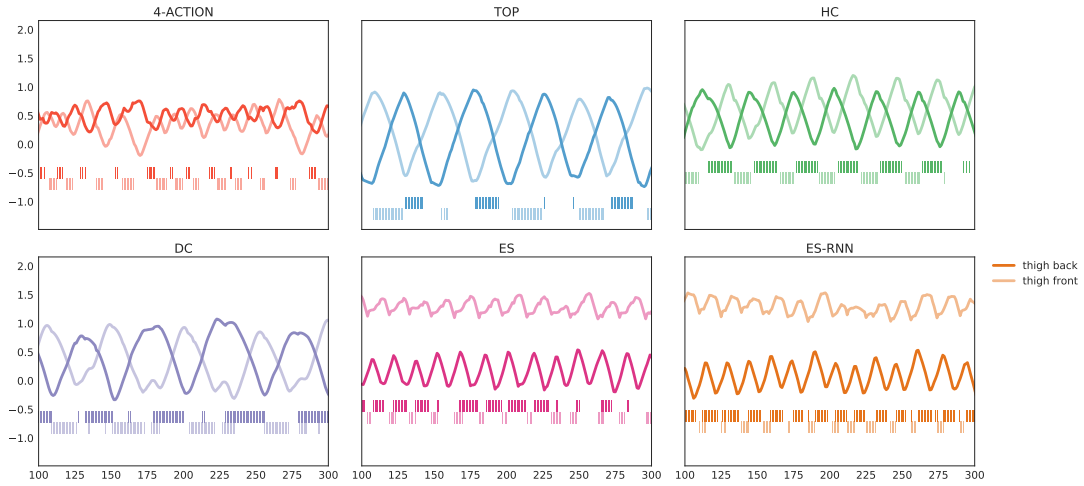


FIGURE 7.3: Phase-shift symmetry results for the best models.

Table of results:

MODEL	ROBUSTNESS	STABILITY			PHASE SHIFT
		VERT.	PITCH.	VEL.	
4-ACTION	3.45	1.12 ± 0.22	0.84 ± 0.049	1.19 ± 0.057	0.63 ± 0.016
TOP	3.84	1.31 ± 0.025	1.29 ± 0.016	1.38 ± 0.027	0.90 ± 0.022
HC	8.35	1.27 ± 0.023	1.24 ± 0.012	1.20 ± 0.022	$\mathbf{0.47 \pm 0.0055}$
DC	**8.44**	0.82 ± 0.035	1.27 ± 0.032	1.10 ± 0.035	0.67 ± 0.013
ES	7.93	1.32 ± 0.014	1.20 ± 0.013	1.22 ± 0.011	3.28 ± 0.0057
ES-RNN	7.32	$\mathbf{0.81 \pm 0.013}$	$\mathbf{1.14 \pm 0.014}$	$\mathbf{0.97 \pm 0.019}$	3.54 ± 0.019

TABLE 7.1: Final comparative results for each model .

7.0.2 Discussion

The results tell us three important things:

1. DDPG produces more human-like gaits than ES. Despite ES having good stability and robustness, the high phase-shift error shows that the algorithm does not produce natural walking policies even though it achieves high *mean return*. Using ES to train a real-life robot from the simple reward function we used would not be effective.
2. DPPG produced the most human-like walking policies when the model architecture used multiple controllers. This implies that instead of adjusting the reward function to produce natural walking gaits, an alternative approach is to introduce interesting model architectures or constraints to force human-like walking behaviour. Additionally, these multi-controller models were much easier to train than the 4-output model.
3. Despite ES-RNN producing the *worst* looking gait, it scored the *highest* on all three steady-state stability metrics. Because the steady-state stability metric relies on recurring dynamics, it makes sense that the recurrent model scored best on this. An interesting idea would be to use recurrent models in the DC and HC models to improve their steady-state stability.

We continue our analysis in the next chapter.

Chapter 8

Conclusion

We start with a reflection on the accomplishment and feasibility of the main goal (see 1 in Chapter 1). Then, we address each of the subgoals outlined at the start of this report and remark on what was gleaned from trying to accomplish the goal. We culminate with some final remarks on the results of the experiments and conclude with a discussion on future work.

8.1 The Main Goal

To reiterate, the *main goal* of this report was the following:

> *Explore whether optimum policies learned by machine learning methods equate to 'natural' and useful policies found in real life, in the context of learning to walk.*

We divided this into two problems:

1. Compare *evolutionary strategies* and *policy gradient methods* (of which we chose *DDPG*).
2. See how well each of the algorithms learned a *robust* and *realistic* walking gait.

The clearest way to represent the comparison of ES and DDPG is via a table of positives and negatives of each:

Evolutionary Strategies

Positives	**Negatives**
• Simple to implement for any problem • Usable on non-differential models • Will converge to a (good) local optimum (for maximum *mean return*)	• Does not learn diverse actions if the reward function is too abstract • Sample inefficient

Deep Deterministic Policy Gradients

Positives	Negatives
• Sample efficient • Learns policies that are correlated to the state space	• Complex hyperparameter tuning • High reliance on initial conditions (e.g. different seeds can cause failure) • May never converge or can completely 'forget' a good policy during training

Arguably, ES and DDPG are both quite similar algorithms. DDPG can be seen as perturbing the output actions of a model to estimate the gradient that maximises the return. ES can be seen as perturbing the parameters of the model to estimate the gradient that maximise the return. It seems that DDPG's reliance on action perturbation is what might make its actions highly correlated with the policy input, hence DDPG learns a richer policy function. In this sense, if the correct hyperparameters are stable, DDPG is the more dominant algorithm. This does raise the question of whether ES can be improved by somehow enforcing action diversity. Something similar to this approach has been attempted by incorporating a maximum-entropy reward on the history of actions [85].

To address the second problem in our main goal, we evaluated the learned walking policies using three walking evaluation metrics: *robustness*, *steady-state stability*, and *phase-shift symmetry*, instead of the common maximum *mean return*. This evaluation metric showed that the 'best' policies learned still produced unrealistic walking gaits. Instead of adjusting the reward function to produce a natural walking gait, we experimented with how different control architectures could potentially produce human-like walking gaits independent of complex reward functions. Although our best walking policies were worse than some of those found in the literature [21] [28] [88], our policies were learned from an abstracted control system and therefore we can claim some success in that sense.

8.2 Evaluating Goals

1. *Learn about deep reinforcement learning as a precursor to implementation.* Prior to starting this report, we had very little experience with reinforcement learning methods and no experience with deep reinforcement learning methods. Immersing ourselves into this promising field was challenging and the troubles encountered while trying to understand and implement the algorithms in this report have given us a greater appreciation for the ideas in the field. The course material in *CS 294: Deep Reinforcement Learning* [39] and *COMPM050: Reinforcement Learning* [65] were especially helpful.

2. *Implement successful walking behaviours in the Bipedal Gym v2 OpenAI Gym environment using deep reinforcement learning, evolutionary strategies, and a baseline method.* The most challenging aspect of this goal was the sensitivity of deep-RL algorithms to initial conditions. A large amount of time was consumed finding stable hyperparameters to solve the walking task.

3. *Evaluate the results on the basis of maximum mean return over time.* Our results showed that evaluating a walk based on the maximum *mean return* will fall victim to *Goodhart's Law*: "When a measure becomes a target, it ceases to be a good measure" [70]. Instead, a model should be evaluated on external metrics that are not the target of the learning system. If we want our agent to learn a walking policy that is human-like, we cannot rely on the model that is only evaluated from the *mean return* over time. This alludes to further work being done on how we evaluate reinforcement learning policies.

4. *Develop metrics to quantitatively analyse a human-like bipedal gait.* The three metrics we used, *robustness*, *steady-state stability*, and *phase shift symmetry*, all provided useful insight. For example, despite the best DDPG model producing a more realistic gait than the best ES model, the best ES model was still more robust to new terrain. This information told us that neither algorithm was producing a realistic gait; this insight would not have been gained if we had purely focused on the *mean return* over time, which ES performed best on.

5. *Explore alterations to the model so as to create a more human-like bipedal gait.* It is possible to 'force' a human-like gait by creating a highly specialised reward function, but this is not an attractive solution and comes with generalisation issues. Instead, we chose to constrain the model the algorithm used to learn the gait in order to allow for more natural walking. When using either a hierarchical or distributed model this resulted in success. This result shows that further research should be done into how the actions of a model are represented, instead of the popular approach to map outputs to actions one-to-one. Inspiration from biology could be useful.

8.3 Future Work

There is a large amount of interest in developing fast, efficient policy-learning algorithms, and for good reason. But, the work in this report has shown that walking policies that are evaluated purely from abstract reward functions are not necessarily useful for real-life applications. A natural extension for future work would be to explore *other* continuous-action control domains where abstract reward functions can be maximised without producing useful policies.

One informative result was the analysis of the responsiveness of ES shown in Figure 5.10 in Chapter 4. Interesting further research would be how to produce richer action responses more like the ones found in Figure 5.11. This would involve utilising the individual actions for the gradient step. Similar to the DDPG model, a 'critic' could be introduced to estimate the return of single actions for an agent; the policy of each individual in the population would then be evaluated on a small sample of states via the 'critic' return estimate. We have performed some preliminary experiments with this approach and it shows potential, but more in-depth work needs to be done to understand this model better.

Additionally, it was clear that the use of recurrent connections in the ES-RNN model caused good steady-state stability scores. As mentioned earlier, it would be interesting to incorporate these recurrent models into the DC and HC model to combine the benefits of each system.

Bibliography

[1] P. Aagaard et al. "Antagonist muscle coactivation during isokinetic knee extension". In: (2000). URL: https://www.ncbi.nlm.nih.gov/pubmed/10755275.

[2] R.M. Alexander. "Bipedal animals and their differences from humans". In: *Journal of Anatomy* 204.5 (2004), pp. 321–330. ISSN: 0021-8782. DOI: doi:10.1111/j.0021-8782.2004.00289.x. URL: https://www.ingentaconnect.com/content/bsc/joa/2004/00000204/00000005/art00002.

[3] alirezamika. *Evostra: A fast Evolution Strategy implementation in Python*.

[4] Marcin Andrychowicz et al. *Hindsight Experience Replay*. 2017. eprint: arXiv:1707.01495. URL: https://arxiv.org/pdf/1707.01495.pdf.

[5] P. Auer, N. Cesa-Bianchi, and P. Fischer. "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.2 (2002), pp. 235–256. ISSN: 1573-0565. DOI: 10.1023/A:1013689704352. URL: https://doi.org/10.1023/A:1013689704352.

[6] Trapit Bansal et al. *Emergent Complexity via Multi-Agent Competition*. 2017. eprint: arXiv:1710.03748. URL: https://arxiv.org/pdf/1710.03748.pdf.

[7] R. Baratta et al. "Muscular coactivation. The role of the antagonist musculature in maintaining knee stability". In: (1988). URL: https://www.ncbi.nlm.nih.gov/pubmed/3377094.

[8] R. Bellman. "The theory of dynamic programming". In: *Bull. Amer. Math. Soc.* 60.6 (Nov. 1954), pp. 503–515. URL: https://projecteuclid.org:443/euclid.bams/1183519147.

[9] Dennis M. Bramble and Daniel E. Lieberman. "Endurance Running and the Evolution of Homo". In: (2004). URL: https://www.nature.com/articles/nature03052.pdf.

[10] Arnaud de Froissard de Broissia and Olivier Sigaud. *Actor-critic versus direct policy search: a comparison based on sample complexity*. 2016. eprint: arXiv:1606.09152. URL: https://arxiv.org/abs/1606.09152.

[11] Dirk Bucher et al. "Central Pattern Generators". In: *eLS*. American Cancer Society, 2015, pp. 1–12. ISBN: 9780470015902. DOI: 10.1002/9780470015902.a0000032.pub2. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470015902.a0000032.pub2. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470015902.a0000032.pub2.

[12] P. Caldirola. "Chronon in Quantum Theory". In: *Lett. Nuovo Cim.* 18 (1977), pp. 465–468. DOI: 10.1007/BF02785060. URL: https://link.springer.com/content/pdf/10.1007%2FBF02785060.pdf.

[13] Matthieu Courbariaux et al. *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*. 2016. eprint: arXiv:1602.02830. URL: https://arxiv.org/abs/1602.02830.

[14] B. Csáji. "Approximation with Artificial Neural Networks". In: (). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.2647&rep=rep1&type=pdf.

[15] G. Cybenko. "Approximation by Superpositions of a Sigmoidal Function". In: (1989). URL: https://pdfs.semanticscholar.org/05ce/b32839c26c8d2cb38d5529cf7720a68c3
pdf.

[16] P. Dayan and G.E. Hinton. *Feudal Reinforcement Learning*. 1993. URL: http://www.cs.toronto.edu/~fritz/absps/dh93.pdf.

[17] D. Dennett. Oxford University Press, 2002.

[18] Yan Duan et al. *Benchmarking Deep Reinforcement Learning for Continuous Control*. 2016. eprint: arXiv:1604.06778. URL: https://arxiv.org/pdf/1604.06778.pdf.

[19] D. Floreano and C. Mattiussi. *Bio-Inspired Artificial Intelligence*. The MIT Press, 2008.

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

[21] T Greijtenbeek, M. van de Panne, and A.F. van der Stappen. "Flexible muscle-based locomotion for bipedal creatures". In: (2013). URL: https://www.bing.com/search?q=flexbile+muscle-ased+lcoomotion+for+bipedal+creatures.

[22] Shixiang Gu et al. *Q-Prop: Sample-Efficient Policy Gradient with An Off-Policy Critic*. 2016. eprint: arXiv:1611.02247. URL: https://arxiv.org/pdf/1611.02247.pdf.

[23] M. Harmon and S. Harmon. *Reinforcement Learning: A Tutorial*. 1996.

[24] H. van Hasselt. "Reinforcement Learning in Continuous State and Action Spaces". In: *Reinforcement Learning: State-of-the-Art*. Ed. by M. Wiering and M. van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 207–251. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_7. URL: https://doi.org/10.1007/978-3-642-27645-3_7.

[25] M. Hausknecht et al. "A Neuroevolution Approach to General Atari Game Playing". In: (2013).

[26] Peter Henderson et al. *Deep Reinforcement Learning that Matters*. 2017. eprint: arXiv:1709.06560. URL: https://arxiv.org/pdf/1709.06560.pdf.

[27] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://doi.org/10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[28] W. Huang, C.M. Chew, and G.S. Hong. "Coordination in CPG and its Application on Bipedal Walking". In: (2008). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4681412&tag=1.

[29] A. Lorincz I. Szita. "Learning Tetris Using the Noisy Cross-Entropy Method". In: (2006). URL: https://pdfs.semanticscholar.org/b199/22afc8678a228c780715d50f5a427d
pdf.

[30] Riashat Islam et al. *Reproducibility of Benchmarked Deep Reinforcement Learning Tasks for Continuous Control*. 2017. eprint: arXiv:1708.04133. URL: https://arxiv.org/abs/1708.04133.

[31] Wiktor A. Janczewski and Jack L. Feldman. "Distinct rhythm generators for inspiration and expiration in the juvenile rat". In: *The Journal of Physiology* 570.2 (), pp. 407–420. DOI: 10.1113/jphysiol.2005.098848. eprint: https://physoc.onlinelibrary.wiley.com/doi/pdf/10.1113/jphysiol.2005.098848. URL: https://physoc.onlinelibrary.wiley.com/doi/abs/10.1113/jphysiol.2005.098848.

[32] M. Jordan. "Attractor Dynamics and Parallelism in a Connectionist Sequential Machine". In: *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*. Hillsdale NJ: Erlbaum, 1986, pp. 531–546.

[33] Michael I. Jordan. "Chapter 25 - Serial Order: A Parallel Distributed Processing Approach". In: *Neural-Network Models of Cognition*. Ed. by John W. Donahoe and Vivian Packard Dorsel. Vol. 121. Advances in Psychology. North-Holland, 1997, pp. 471 –495. DOI: https://doi.org/10.1016/S0166-4115(97)80111-2. URL: http://www.sciencedirect.com/science/article/pii/S0166411597801112.

[34] O Kiehn and O Kjaerulff. "Distribution of Central Pattern Generators for Rhythmic Motor Outputs in the Spinal Cord of Limbed Vertebratesa". In: *Annals of the New York Academy of Sciences* 860.1 (), pp. 110–129. DOI: 10.1111/j.1749-6632.1998.tb09043.x. eprint: https://nyaspubs.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1749-6632.1998.tb09043.x. URL: https://nyaspubs.onlinelibrary.wiley.com/doi/abs/10.1111/j.1749-6632.1998.tb09043.x.

[35] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. eprint: arXiv:1412.6980. URL: https://arxiv.org/abs/1412.6980.

[36] Peter Le et al. "A review of methods to assess coactivation in the spine". In: *Journal of Electromyography and Kinesiology* 32 (2017), pp. 51 –60. ISSN: 1050-6411. DOI: https://doi.org/10.1016/j.jelekin.2016.12.004. URL: http://www.sciencedirect.com/science/article/pii/S1050641116300943.

[37] Joel Lehman et al. *The Surprising Creativity of Digital Evolution: A Collection of Anecdotes from the Evolutionary Computation and Artificial Life Research Communities*. 2018. eprint: arXiv:1803.03453.

[38] Sergey Levine. *Actor-Critic Algorithms - CS294: Deep Reinforcement Learning*. 2017. URL: http://rll.berkeley.edu/deeprlcourse/.

[39] Sergey Levine. *CS294: Deep Reinforcement Learning*. 2017. URL: http://rll.berkeley.edu/deeprlcourse/.

[40] Timothy P. Lillicrap et al. *Continuous control with deep reinforcement learning*. 2015. eprint: arXiv:1509.02971. URL: https://arxiv.org/abs/1509.02971.

[41] Warren P. Lombard and F. M. Abbott. "THE MECHANICAL EFFECTS PRODUCED BY THE CONTRACTION OF INDIVIDUAL MUSCLES OF THE THIGH OF THE FROG". In: *American Journal of Physiology-Legacy Content* 20.1 (1907), pp. 1–60. DOI: 10.1152/ajplegacy.1907.20.1.1. eprint: https://doi.org/10.1152/ajplegacy.1907.20.1.1. URL: https://doi.org/10.1152/ajplegacy.1907.20.1.1.

[42] L. Lundy-Ekman. *Neuroscience: Fundamentals for Rehabilitaion*. Elsevier Helath Sciences.

[43] S. Mannor, R. Rubinstein, and Y. Gat. *The Cross Entropy method for Fast Policy Search*. 2003. URL: http://www.aaai.org/Papers/ICML/2003/ICML03-068.pdf.

[44] I. Menache, S. Mannor, and N. Shimkin. "Basis Function Adaptation in Temporal Difference Reinforcement Learning". In: *Annals of Operations Research* 134.1 (2005), pp. 215–238. ISSN: 1572-9338. DOI: 10.1007/s10479-005-5732-z. URL: https://doi.org/10.1007/s10479-005-5732-z.

[45] V. Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. URL: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf.

[46] Yael Niv. "Reinforcement learning in the brain". In: (2009). URL: https://www.princeton.edu/~yael/Publications/Niv2009.pdf.

[47] M. Olivares-Mendez et al. "Cross-Entropy Optimization for Scaling Factors of a Fuzzy Controller: A See-and-Avoid Approach for Unmanned Aerial Systems". In: *Journal of Intelligent & Robotic Systems* 69.1 (2013), pp. 189–205. ISSN:

1573-0409. DOI: 10.1007/s10846-012-9791-5. URL: https://doi.org/10.1007/s10846-012-9791-5.

[48] OpenAI. *OpenAI Gym: BipedalWalker-v2 Environment*. URL: BipedalWalker-v2.

[49] Mikhail Pavlov, Sergey Kolesnikov, and Sergey M. Plis. *Run skeleton run: skeletal model in a physics-based simulation*. 2017. eprint: arXiv:1711.06922. URL: https://arxiv.org/pdf/1711.06922.pdf.

[50] Xue Bin Peng et al. "DeepLoco: Dynamic Locomotion Skills Using Hierarchical Deep Reinforcement Learning". In: *ACM Transactions on Graphics (Proc. SIGGRAPH 2017)* 36.4 (2017).

[51] M. Pfeiffer. "Reinforcement learning of strategies for Settlers of Catan". In: (Mar. 2018). URL: https://www.researchgate.net/publication/228728063_Reinforcement_learning_of_strategies_for_Settlers_of_Catan.

[52] Ioannis Poulakakis et al. "Chapter 7 - Legged Robots with Bioinspired Morphology". In: *Bioinspired Legged Locomotion*. Ed. by Sharbafi et al. Butterworth-Heinemann, 2017, pp. 457 –561. ISBN: 978-0-12-803766-9. DOI: https://doi.org/10.1016/B978-0-12-803766-9.00010-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128037669000105.

[53] D Prafulla et al. *OpenAI Baselines*. https://github.com/openai/baselines. 2017.

[54] M. Raibert. *Legged Robots That Balance*. MIT Press.

[55] I. Rechenber. "Evolutionsstrategien". In: *In Simulationsmethoden in def Medizin und Biologie* (1978), pp. 83–114.

[56] Graham Richards. "Freed hands or enslaved feet? A note on the behavioural implications of ground-dwelling bipedalism". In: *Journal of Human Evolution* 15.3 (1986), pp. 143 –150. ISSN: 0047-2484. DOI: https://doi.org/10.1016/S0047-2484(86)80041-0. URL: http://www.sciencedirect.com/science/article/pii/S0047248486800410.

[57] D. Rumelhard, G. Hinton, and R. Williams. "Learning internal representation by error propagation". In: *Parallel Distributed Processing - Volume 1: Foundation* (1986).

[58] D. Rumelhard, G. Hinton, and R. Williams. *Parallel distributed processing: Explorations in the microstructure of cognition - Volume 1: Foundation*. MIT Press Cambridge Mass., 1986.

[59] Carl Sagan. *The Dragons of Eden*. Random House, 1977.

[60] Tim Salimans et al. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. 2017. eprint: arXiv:1703.03864. URL: https://arxiv.org/abs/1703.03864.

[61] S. Sarjant et al. "Using the online cross-entropy method to learn relational policies for playing different games". In: *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*. 2011, pp. 182–189. DOI: 10.1109/CIG.2011.6032005.

[62] D. Silver et al. *Mastering the game of Go without human knowledge*. 2017. URL: https://www.nature.com/articles/nature24270.

[63] David Silver. 2015. URL: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/intro_RL.pdf.

[64] David Silver. 2015. URL: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/DP.pdf.

[65] David Silver. *COMPM050: Reinforcement Learning*. 2015. URL: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html.

[66] David Silver et al. "Deterministic Policy Gradient Algorithms". In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*. ICML'14. Beijing, China: JMLR.org, 2014, pp. I–387–I–395. URL: http://dl.acm.org/citation.cfm?id=3044805.3044850.

[67] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. eprint: arXiv:1712.01815. URL: https://arxiv.org/abs/1712.01815.

[68] W. D. Smart and L. Pack Kaelbling. "Effective reinforcement learning for mobile robots". In: *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*. Vol. 4. 2002, 3404–3410 vol.4. DOI: 10.1109/ROBOT.2002.1014237. URL: http://ieeexplore.ieee.org/document/1014237/?anchor=citations.

[69] Richard C. Snyder. "Bipedal Locomotion of the Lizard Basiliscus basiliscus". In: *Copeia* 1949.2 (1949), pp. 129–137. URL: http://www.jstor.org/stable/1438487.

[70] M. Strathern. "'Improving ratings':audit in the British University system". In: (1997). URL: doi:10.1002/(SICI)1234-981X(199707)5:33.0.CO;2-4.

[71] Felipe Petroski Such et al. *Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning*. 2017. eprint: arXiv:1712.06567.

[72] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. 1st. Cambridge, MA, USA: MIT Press, 2018. ISBN: 0262193981. URL: http://incompleteideas.net/book/bookdraft2018feb28.pdf.

[73] R.S. Sutton et al. *Policy Gradient Methods for Reinforcement Learning with Function Approximation*. 2000.

[74] G. Tesauro. "Temporal Difference Learning and TD-Gammon". In: *Communications of the ACM* (1995). URL: http://www.bkgm.com/articles/tesauro/tdl.html.

[75] Tondu and Bertrand. *Artificial Muscles for Humanoid Robots*. 2007.

[76] George Tucker et al. *The Mirage of Action-Dependent Baselines in Reinforcement Learning*. 2018. eprint: arXiv:1802.10031. URL: https://arxiv.org/abs/1802.10031.

[77] G.E. Uhlenbeck and L.S. Ornstein. "On the Theory of the Brownian Motion". In: *Phys. Rev.* 36 (5 1930), pp. 823–841. DOI: 10.1103/PhysRev.36.823. URL: https://link.aps.org/doi/10.1103/PhysRev.36.823.

[78] Alexander Sasha Vezhnevets et al. *FeUdal Networks for Hierarchical Reinforcement Learning*. 2017. eprint: arXiv:1703.01161. URL: https://arxiv.org/abs/1703.01161.

[79] Paweł Wawrzyński. "Control Policy with Autocorrelated Noise in Reinforcement Learning for Robotics". In: *International Journal of Machine Learning and Computing, Vol. 5, No. 2* (2015). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.696.1149&rep=rep1&type=pdf.

[80] P. J. Werbos. "Neural networks and the human mind: new mathematics fits humanistic insight". In: *[Proceedings] 1992 IEEE International Conference on Systems Man and Cybernetics*. 1992, 78–83 vol.1. DOI: 10.1109/ICSMC.1992.271798.

[81] S. Whiteson et al. "Protecting against evaluation overfitting in empirical reinforcement learning". In: *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*. 2011, pp. 120–127. DOI: 10.1109/ADPRL.2011.5967363.

[82] D. Wiestra et al. "Natural Evolution Strategies". In: *Journal of Machine Learning Research* (2014).

[83] R.J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning* 8.3 (1992), pp. 229–256. ISSN: 1573-0565. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[84] R.A. Wise. "Neuroleptics and operant behavior: The anhedonia hypothesis". In: *Behavioral and Brain Sciences* 5.1 (1982), 39–53. DOI: 10.1017/S0140525X00010372. URL: https://www.cambridge.org/core/services/aop-cambridge-core/content/view/EAF66A9C61B9BDD03D35C72E2077620D/S0140525X00010372a.pdf/neuroleptics_and_operant_behavior_the_anhedonia_hypothesis.pdf.

[85] A. D. Wissner-Gross and C. E. Freer. "Causal Entropic Forces". In: *Phys. Rev. Lett.* 110 (16 2013), p. 168702. DOI: 10.1103/PhysRevLett.110.168702. URL: https://link.aps.org/doi/10.1103/PhysRevLett.110.168702.

[86] Kelvin Xu et al. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*. 2015. eprint: arXiv:1502.03044. URL: https://arxiv.org/abs/1502.03044.

[87] E. Yazdi and A.T. Haghighat. "Evolution of Biped Walking Using Neural Oscillators Controller and Harmony Search Algorithm Optimizer". In: (2010). URL: https://pdfs.semanticscholar.org/0ea0/96611d851de2b703db1794a5bf17ace007d0.pdf.

[88] J. Zhang, X. Zhao, and C. Qi. "A Series Inspired CPG Model for Robot Walking Control". In: *2012 11th International Conference on Machine Learning and Applications*. Vol. 1. 2012, pp. 444–447.