Neural Network Driven Locomotion of General Bodies

Using Reinforcement Learning and Evolutionary Algorithms

Master's thesis in Complex Adaptive Systems

Jonathan Kammerland
Ulf Hjohlman

Department of Physics
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018

Neural Network Driven Locomotion of General Bodies

Using Reinforcement Learning and Evolutionary Algorithms

Jonathan Kammerland
Ulf Hjohlman

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Division of Complex Adaptive Systems
Chalmers University of Technology
Gothenburg, Sweden 2018

Neural Network Driven Locomotion of General Bodies
Using Reinforcement Learning and Evolutionary Algorithms
Jonathan Kammerland
Ulf Hjohlman

Supervisor & Examiner: Kristian Gustafsson, Department of Physics

Master's Thesis 2018
Department of Physics
Division of Complex Adaptive Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A Quadruped robot simulated in the MuJoCo physics engine.

Typeset in LaTeX
Gothenburg, Sweden 2018

Neural Net Driven Locomotion of General Bodies
Using Reinforcement Learning and Evolutionary Algorithms
Jonathan Kammerland
Ulf Hjohlman
Department of Physics
Chalmers University of Technology

Abstract

Producing locomotion controllers for general bodies has multiple applications, foremost in robotics. Most robot controllers are today developed through the use of autonomous control, trying to deviated as little as possible from some target gait. This target is usually supplied in the form of motion capture data or determined by a human. This work attempts to instead use machine learning methods to allow simulated agents to learn locomotive behaviour. The environment is void of any preconceived notions regarding what the resulting gait should look like, and simply encourages the agents to maintain forward velocity.

The controllers are represented in the form of neural networks mapping sensory inputs to actuator outputs. Three machine learning methods are explored for training these networks; evolutionary algorithms with niches, the reinforcement learning algorithm Proximal Policy Optimization, and a novel algorithm combining the two previous methods. The hybrid algorithm out-performs the rest. It works well on simpler problems such as a 2-dimensional biped, but struggles on the difficult unstable 3-dimensional problem of a full humanoid robot.

The results are not competitive with traditional autonomous control results when controlling well understood bodies like humanoids. However the algorithm does succeed in finding creative gaits for less structured problems and odd bodies, something that a human might find difficult.

Keywords: locomotion, walking, AI, RL, gait, PPO, EA, MuJoCo.

Acknowledgements

Thanks to Kristian for supervising and putting up with us. Thanks to everyone who worked on something listed in the bibliography. Thanks to CSN, and by extension the taxpayers of Sweden for funding us. Also thanks to David Frisk for this LaTeX template.

Jonathan Kammerland & Ulf Hjohlman, Gothenburg, January 2018

Contents

1
Introduction

Locomotion is a fundamental necessity for many living beings in our spatially defined world. With advancements in robotics and artificial intelligence the task of deriving locomotion for artificial use is also of growing importance, for example see the DURUS robot [1]. In many ways a gait[1] can be seen as the solution to an optimization problem. What gait is optimal depends on the body structure, but also on the desired speed, energy efficiency, and balance required by its enactor. A horse for example can move in a walk, trot, canter, gallop, or rack. All are interesting stable gaits suited for different situations. Environmental factors such as terrain fluctuations, friction, and type of obstacles encountered also play a role in determining what can be deemed optimal movement.

In a biological setting locomotion is an essential tool for survival and it is no wonder that animals have developed efficient locomotion through the process of natural selection. However in robotics and virtually simulated agents how to go about this optimization is not straightforward. On an abstract level the problem is to map sensory inputs to motor and muscle outputs through the use of some controller. The sensors will encompass information such as the bodies current limb positions and velocities as well as environmental information through touch and range sensors. The number of sensory inputs grows rather quickly with the complexity of the body, and the controller needs to be general enough such that it can be adapted to these high dimensional inputs and outputs. It is also important to note that nothing is assumed about the solution gait. Other approaches might use keyframes or other data and train a robot to follow a predetermined gait with more traditional autonomous control [2, 3]. In the case of the problem discussed in this work there is no such error signal to follow. The movement is derived in a more organic way as the result of optimization for a higher level goal, such as forward velocity.

Efficient solutions to this problem would allow for direct implementations within robotics. There are also applications for the animation of characters in video games and movies [2]. Especially for characters with more exotic body structures where a human does not have much intuition for how they would move in a natural manner, such as fantasy creatures and monsters etc. Additionally it could grant insight into how alterations in environmental factors would affect locomotion. How does a human or machine best traverse Mars' rigid landscape with only a third of the gravitational force she is used to? Perhaps the intuition learned by a lifetime of

[1] locomotion pattern

living on Earth is not optimal for the task. Improving our understanding of how alterations in bodies affect movement is also of interest in medicine and sports [4].

One potential way of representing a controller is through the use of an artificial neural network. The use of neural networks (NNs) for continuous control tasks is an active field of research [5, 6, 7, 8, 9, 10, 11] with many breakthroughs in recent years due to both increases in computational power and algorithmic advancements. NNs have proven to be exceptionally useful for finding patterns in high-dimensional data, and can be a powerful tool. However, the difficulty lies in training them, i.e. tweaking their internal parameters in a systematic way to achieve the desired behavior.

One way of training neural networks is through the stochastic optimization method known as evolutionary algorithms, EAs [12, 13]. Definitions and nomenclature differs in the litterature, but in this work EAs involve the following steps; 1) maintaining a population of candidate solutions, 2) altering the solutions through the use of different operators, and 3) a selection process favoring 'better' solutions according to some fitness measure, while discarding worse ones.

Another method of training NNs is through the use of reinforcement learning [10, 11]. The idea is to let agents freely explore a policy space[2], and reward the desired behaviour. This guides the agents to learn by themselves as the algorithms controlling them try to maximize the reward they receive. The last few years' algorithmic advancements and increases in computational power have lead to a number of breakthroughs in RL [14, 15], and RL already boasts state of the art performance when it comes to locomotion control [16, 17, 18].

Our work examines methods for developing locomotion skills in agents so that they can traverse some environment. By studying previous models and incorporating novel ideas we hope to achieve higher performance and a greater understanding of the ability to move under the laws of physics. In addition this will also grant us further insight into the workings of these optimization methods, which are applicable to a broad range of tasks [15, 14].

The next chapter introduces some theory behind evolutionary algorithms and reinforcement learning. This paves the way for Chapter 3 where this theory is applied more directly to the context of locomotion, along with an overview of previous work on the subject. In Chapter 4 we present the algorithms in the form of more concrete pseudo-code, and introduce a hybrid method combining EA and RL. Chapter 5 presents the performance of the algorithms on a few case studies, and Chapter 6 discusses these results in a larger context to draw some conclusions.

[2] A policy can be thought of as the strategy governing an agents actions given measurements of its environment

2

Machine Learning Theory

In the following sections general theory about the algorithms and models are presented, with more problem-specific details to come in the subsequent chapters. A short description of what each subsection contains:

- **Neural network** theory and models. An introduction to the concept and dynamics of neural networks we employ. How they can be used as controllers for mapping sensor states to actuator activation.
- **Evolutionary algorithms** for developing synaptic weights (and topology) of a neural network. Evolutionary operators such as crossover, mutation, replacement, speciation for the direct encoding scheme of a neural network are introduced.
- **Reinforcement learning** theory and how to train a neural network's synaptic weights in a continuous state and action space with gradient descent methods.

2.1 Neural Networks

Central nervous systems control movement in virtually all animal life. From some external stimuli of receptors, electrical signals are fed to a neural network which can process the information. Electrical impulses from the neural network can then activate motor cells or muscles to do useful work. Therefore it can be argued that trying to model this biological system for agents that need sophisticated locomotion abilities might be eligible. Models used for computational purposes are called artificial neural networks.

The artificial neural network (ANN), simply called neural network, is inspired by the biological neural network. The biological neural network is a set of many individual but connected computational units called neurons, see Figure 2.1. The connective links between neurons are called *synapses* and are characterized by their synaptic strength. Stimulation of the synapses either strengthen or weaken them over time [19]. This adaption in synapses is a concept referred to as *neural plasticity* or *synaptic plasticity* which the Hebbian theory tries to explain [20]. Synapses are often referred to as *synaptic weights* or just *weights* as they appear to influence the spiking activity between connected neurons [21]. How the exact dynamics work still remains an open question though.

Now let us introduce a model for the neuron used in our neural networks. There are many models that can be used to describe a neuron. However the most common artificial neuron model used in machine learning are variants of the McCulloch–Pitts model [22]. The McCulloch–Pitts neuron operates in discrete time where the inputs

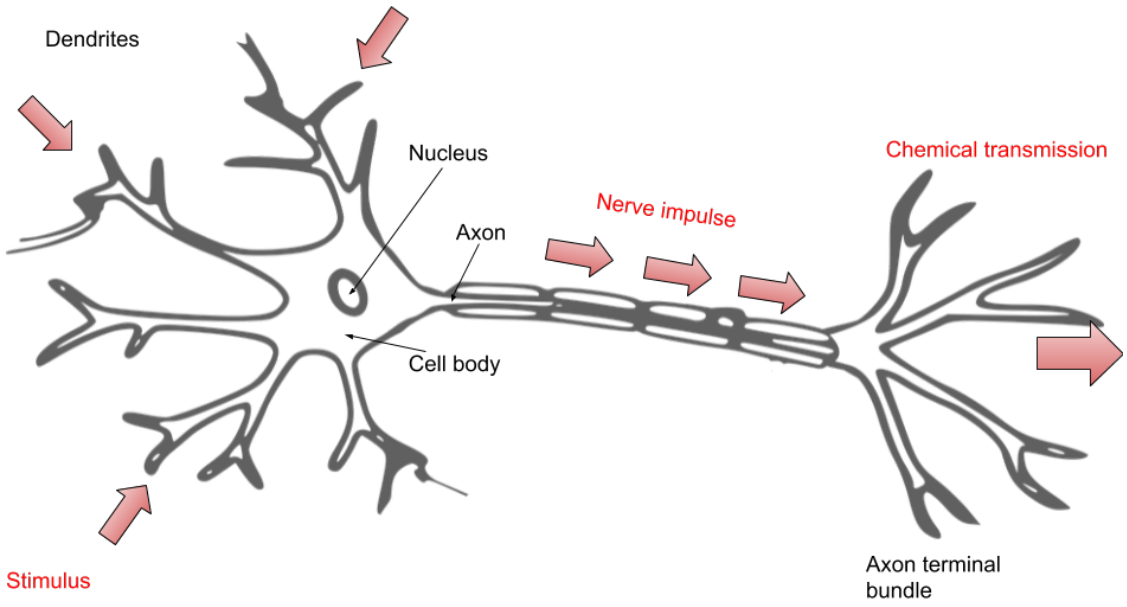

Figure 2.1: A biological model of a neuron. The dendrites receives stimulus from other neurons through chemical transmitters. A nerve impulse can be generated by the cell body in response to its stimulus. The generated nerve impulse is then carried by the axon which connects to other cells or dendrites at the axon terminal bundle.

are summed in a weighted manner and the result is fed to a activation function, see Figure 2.2. The model is used to perform a non-linear mapping

$$\text{Neuron} : \mathbb{R}^N \to \mathbb{R}$$

of N inputs to a scalar output. The mathematical description of the neuron in Figure 2.2 is

$$u = \sum_{i=1}^{N} w_i x_i + b$$
$$y = \varphi(u)$$

where y is the output, u the net input, φ the activation function, w_i the weight of connection i, x_i the input i and b is the bias of the neuron. A more convenient way is to represent the sum operation in vector form so that the input is represented as $\boldsymbol{x} = [x_1, x_2, ..., x_N, 1]^T$ and the weights as $\boldsymbol{w} = [w_1, w_2, ..., w_N, b]^T$ by appending the bias term. The net input u is then

$$u = \boldsymbol{w}^T \boldsymbol{x} = \boldsymbol{w} \cdot \boldsymbol{x}$$

The non-linear properties of the model into play from the activation function. Standard activation functions include sigmoidial functions or S-shaped functions [23]. We use

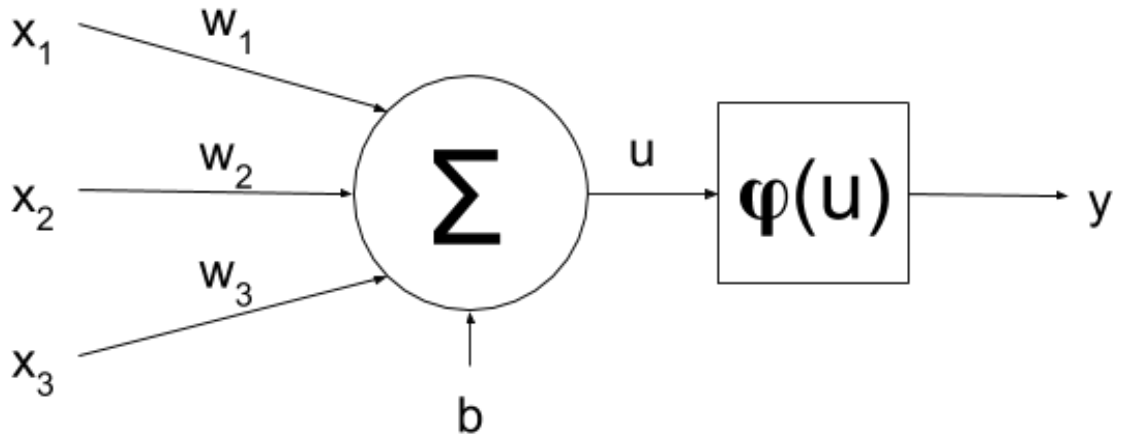

Figure 2.2: A non-linear model of a neuron. The activation level y of the neuron is calculated from some inputs x_i. The incoming connections with weights w_i to the summation junction Σ computes the net input u as a weighted sum of the inputs together with the bias b. Then the net input u is fed to a non-linear activation function φ.

$$\varphi(u) = tanh(u) = \frac{1}{2}\frac{e^u - e^{-u}}{e^u + e^{-u}} \tag{2.1}$$

The function above essentially "squashes" the input into a scalar value in the range $(-1, 1)$. The point of using functions such as $tanh(u)$, except being non-linear, is that they are *continuous* and *smooth*, thus *differentiable*. Being differentiable is necessary when "training" weights with *gradient descent* which will be introduced later. But why use activation functions at all? Without non-linear activation functions, the computation from a set of neurons is just a composition of linear functions, which is also a linear function. This would limit the neural networks modelling power to that of a linear system. With non-linearity embedded however, the networks can act as *universal approximators* which will be discussed later.

Forming a network $\boldsymbol{\Phi} = \boldsymbol{\Phi}(\boldsymbol{x}; \boldsymbol{w})$, from multiple neurons, connected by the weights $\boldsymbol{w}$ and taking input $\boldsymbol{x}$, can be viewed as a function performing the mapping

$$\Phi : \mathbb{R}^N \to \mathbb{R}^m$$

Here N is the number of inputs, i.e $N = dim(\boldsymbol{x})$, and m is the number of outputs. While the network can have arbitrary *architecture*, we use the most common type of all, the multilayer perceptron [22]. This kind of architecture has its neurons grouped into layers such that there are only connections between adjacent layers, see Figure 2.3. As an example, the computation of layer l in a feed forward neural network would be

$$y_j^{(l)} = \varphi(u_j^{(l)}) = \varphi\left(\sum_{i=1}^{N_{l-1}+1} w_{ji}^{(l-1)} y_i^{(l-1)}\right) \quad \Leftrightarrow$$
$$\boldsymbol{y}^{(l)} = \varphi(\boldsymbol{u}^{(l)}) = \varphi(\boldsymbol{W}^{(l-1)}\boldsymbol{y}^{(l-1)})$$

where y_j is the output of neuron j, and w_{ji}^{l-1}(element at row j and column i in matrix $\boldsymbol{W}^{l-1}$) is the connection from neuron i in layer $(l-1)$ to neuron j in layer l. Here we have appended the bias term into the weight matrix such that we sum to $N_{l-1}+1$ if N_{l-1} is the number of neurons in layer $(l-1)$, so that $y_{N+1}^{(l-1)} = 1$ multiplies with the bias weight. To get the full computation of a network $\boldsymbol{\Phi}(\boldsymbol{x}; \boldsymbol{w})$ we just have to recursively expand the equations above until we reach layer 0, taking $\boldsymbol{y}^{(0)} = \boldsymbol{x}$. In this case, $\boldsymbol{x}$ is a vector of our sensor signals at any particular time from different parts of the body we are trying to control.
However the question remains whether the neural network can learn anything useful in terms of producing locomotion in bodies. It can be shown that a set of connected neurons with non-linear activation functions are *universal function approximators* [22, 23]. That is, a neural network can approximate an input-output mapping for any continuous function. There is also several derivations of the *universal dynamical system approximation theorem* [22, 24] for *recurrent neural networks*, that has an evolving network state in time, see the *discrete-time recurrent neural network* in Figure 6.2. For example, if there exists a non-linear dynamical system which produce a functioning gait in an agent given some sensory inputs, then it can be approximated with a recurrent neural network. This raises the question how good the approximation can be? Indeed both the universal approximation theorems states that the approximation can be made to any degree of accuracy by just adding more neurons to the network.
We focus on the feed forward kind in this project like many others for neural network driven bodies [10, 25] due to their simplicity. Examples of using recurrent neural network models can be seen in Ref. [8, 26]. However we do some testing with a recurrent networks of the same form as in Figure 2.3. The recurrent network with one hidden global feedback layer is sometimes referred to as a simple recurrent network.
To train neural networks to produce some desired response to a input one can use gradient descent to modify the weights. This is the case in reinforcement learning, where we explore desirable actions given some state and reinforce those behaviours by adjusting the weights accordingly. This involves setting up a function of the error between the output and the desired response. Then minimizing the error function with respect to the weights. We use a commonly used error function [22, p. 157], denoted E, given by

$$E(\boldsymbol{x}; \boldsymbol{w}) = \sum_j (d_j - y_j)^2$$

where d_j is the desired output of neuron j and y_j is the actual output when feeding $\boldsymbol{x}$ as input to the network. Without going into further details, the gradient descent procedure to update the weights to minimize the error E goes as follows

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) - \eta \frac{\partial E}{\partial \boldsymbol{w}}$$

where n is the current iteration, η the step size in the descent direction $-\frac{\partial E}{\partial \boldsymbol{w}}$. By presenting randomly chosen input-output mappings, $\boldsymbol{x}$ to $\boldsymbol{d}$, we can adjust the weights a small amount every iteration in order to minimize the error over many iterations for all mappings. This is commonly known as stochastic gradient descent (SGD) [22, 27, 13].

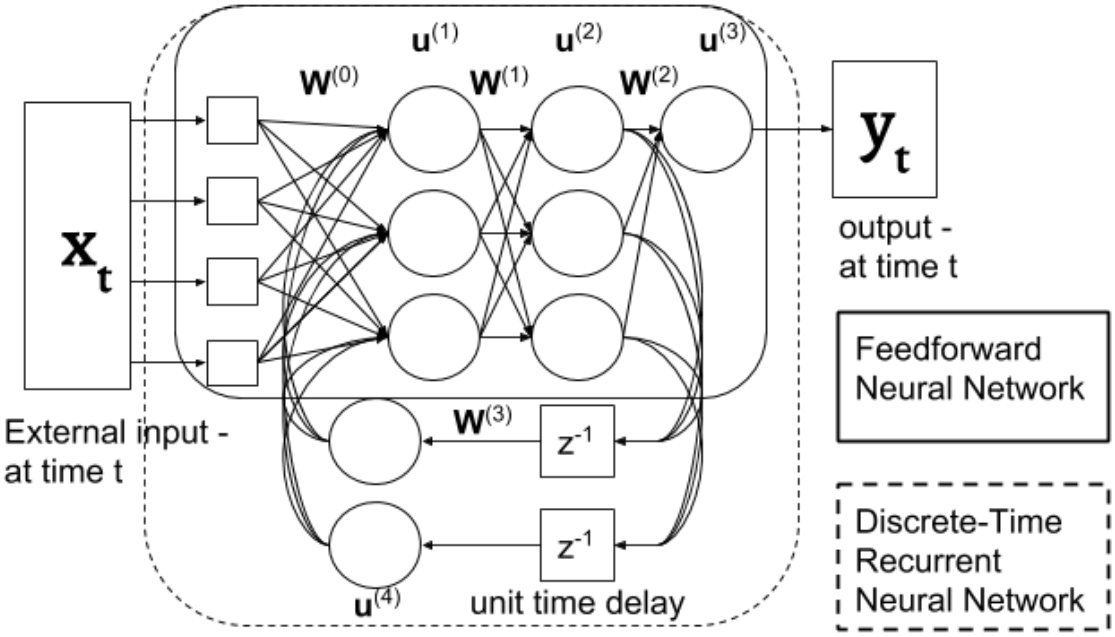

Figure 2.3: Standard neural network models. The circles are the individual neurons grouped into layers. The neurons calculate their activation level from the net input $\boldsymbol{u}^{(l)}$. The arrows are the networks weighted connections between layers denoted $\boldsymbol{W}^{(l)}$. The boxes just passes input onward without any changes, however the boxes with z^{-1} delays the propagation of the input by one time-step. The solid lines encircles a feed forward neural network (FFNN), where all connection feed signals strictly forward making the network time-invariant. The extension inside the dashed lines is a recurrent neural network that has time dependency.

2.2 Evolutionary Algorithms

Evolutionary algorithms (EAs) is a class of stochastic algorithms that is inspired by biological evolution. It involves taking a set of parameters to a given problem, trying different combinations of those parameters and randomly exploring what works better. In our case we wish to evolve neural networks for controlling various bodies. If we exclude the architectural degrees of freedom for a neural network, the parameters of a neural network are it's weights, $\boldsymbol{w}$, see Section 2.1.
The evolutionary process of the population operates via selection and survival of the fittest. With this method the more fit solutions, as judged by an objective function,

have a higher chance of reproduction and survival. There are multiple operators in EAs which control the search, concentrating it to regions centered around promising candidate solutions. The operators and components to be introduced are

- **Encoding** method that maps a solution to a representation which the applied operators can modify. For example binary [12, p. 40-41], gray [28] or floating point encoding [13].
- **Objective function** to evaluate *fitness* or solution performance.
- **Selection operator** which applies evolutionary pressure to a population. Often used for selection when recombining solutions (referred to as *crossover*) and replacement of old individuals.
- **Crossover operator** which recombines the encoded genetic material from two or more individuals.
- **Mutation operator** that adds or introduces new traits to a population, allowing it to search globally in the search space.
- **Replacement operator** that determines how generations interact and/or allows for more evolutionary pressure.

Convergence rate and performance of the algorithm is highly dependent on the selective pressure and mutation rate, which are controlled by a set of hyper-parameters. Unfortunately, optimal parameters in the sense that the algorithm does not suffer from *premature convergence* or very slow convergence, are often problem dependent. Premature convergence often comes from lack of diversity of the population, a result of poor parameters, such as low mutation rates and high selective pressure, or difficult fitness landscape with hard-to-escape-from local optima. More advanced operators have been developed to help mitigate these problems, such as *speciation* which restricts mating and competition between individuals.
Applying EAs to neural networks is often referred to as *neuroevolution*. Where weights-, network structure- and learning rules can be evolved based on the operators mentioned above. The following sections describe the evolutionary operators used in this project.

2.2.1 Genetic Encoding

The most common forms to encode neural networks are either direct or indirect encoding. In short

- **Direct encoding** means that all weights and parameters defining the network directly represent the individual genes. In the case of all genes being floating point values, a genome is simply encoded into a vector $\boldsymbol{x} \in \mathbb{R}^N$ with N genes. This allows for efficient modification with simpler genetic operators such as *creep mutations* [12] and *averaging crossover* [13] as these operators only perform simple arithmetic on individual genes.
- **Indirect encoding** encodes all parameters for example into binary or text strings. This is useful for when you want to explore solutions of arbitrary complexity such as evolving neural network architectures. Because this method allows you to represent and manipulate complex structures more easily.

An example of successful algorithms evolving neural networks is *neuro-evolution of augmenting topologies* (NEAT) [26] and its derivatives such as HyperNEAT [29]

where both network topology and synaptic weights are evolved simultaneously in a indirect encoding scheme. For this application, all parameters of the network are encoded directly into a vector denoted $\boldsymbol{w}$.

2.2.2 Crossover operators

The crossover operator combines the genome from two or more individuals in order to create offspring. Variants include one-point, two-point, n-point crossover [13, p. 144-146], where the genome is cut and recombined in chunks. In the *messy* variant the cut genome sequences are recombined with no restrictions on ordering, allowing for genomes of variable length [13, p.159-160].
With fixed genome size, an arithmetic crossover operator for the floating point encoding can be done in directional fashion, searching between two parents genes (x_{1j}, x_{2j}) where j is the gene index as

$$x_{\mu j} = (1 - \gamma)x_{2j} + \gamma x_{1j} \tag{2.2}$$

where $\gamma \sim U(0, 1)$ is a uniformly distributed variable in the range $[0, 1]$, sampled for each gene j in an offspring μ. Other options for this case include weighting the search towards the more fit parent. Extended to multiple parents one can describe this as

$$x_{\mu j} = \sum_{i=1}^{n} \alpha_i x_{ij} \tag{2.3}$$

where n is the number of parents, α_i the weight factor and $\sum_{i=1}^{n} \alpha_i = 1$ such that offspring $\boldsymbol{x}_\mu$ is the weighted average of the parents. Both methods above are described in Ref. [13].
Note that equation (2.3) is of central role in the evolutionary algorithm *evolutionary strategies* (ES), where the whole population is replaced by said operator after mutation is applied to each individual. This method searches the parameter space in a swarm around the weighted average of the current population.

2.2.3 Mutation operators

The mutation operator simulates the random replication errors in cell proliferation or external perturbations causing damage to the genome such as radiation. For neural networks, mutation operations can include adding or deleting neurons and connections, as well as modifying weights, hyper parameters and activation functions.
A normal operator for mutating network weights is using *creep mutations* [12, p. 54] which applies a small Gaussian perturbation δ_j to the weight according to

$$\hat{x}_{\mu j} = x_{\mu j} + \delta_j, \quad \delta_j \sim N(m_j, \sigma_j) \tag{2.4}$$

where $\hat{x}_{\mu j}$ is the updated weight j for individual $\boldsymbol{x}_\mu$, perturbed by a normal distributed variable with variance σ_j^2 and mean m_j (often set to zero). Note that variances associated with δ_j are often set to the same fixed value σ. Momentum can

be used to accelerate training time or help with exploration [30]. The momentum m_j of a gene j can be done with decay and gain as in

$$\hat{m}_j = \gamma m_j + \eta\delta_j, \quad \eta > 0, \quad \gamma \in [0, 1) \tag{2.5}$$

where γ is the decay factor, η is the momentum gain and $\hat{m}_j$ is the updated momentum. The absolute value of the momentum m_j is then clipped to a fraction of the variance σ_j.
Another approach is to use gradient based mutations, called *safe mutations* by Ref. [25]. This method perturb the weight vector such that it does not change the input-output mapping too much. With the reasoning being to not render the individual very unfit due to major behavioral changes. This can be done by investigating how much the neural networks output diverges from a given perturbation $\boldsymbol{\Delta w}$, in response to its inputs from a previous simulation. If we let $\boldsymbol{x}_i$ denote the input vector (all sensor values at time step i) and $\boldsymbol{w}$ be the weights of network $\boldsymbol{\Phi}$, the output is $\boldsymbol{y}_i = \boldsymbol{\Phi}(\boldsymbol{x}_i; \boldsymbol{w})$ or $y_{ik} = \Phi(\boldsymbol{x}_i; \boldsymbol{w})_k$ for the kth output as in Section 2.1. In Ref. [25] they define the divergence D of perturbation $\Delta\boldsymbol{w}$ as

$$D(\boldsymbol{\Delta w}; \boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{m}\Big[\Phi(\boldsymbol{x}_i; \boldsymbol{w})_k - \Phi(\boldsymbol{x}_i; \boldsymbol{w} + \boldsymbol{\Delta w})_k\Big]^2 \tag{2.6}$$

where m is the number of outputs and N the number of samples taken in order to estimate the divergence. Taking a random direction $\boldsymbol{d}$ and step size η we can form a perturbation $\boldsymbol{\Delta w} = \eta\boldsymbol{d}$ of the weight vector. In order to produce a divergence D below some threshold we sample a random direction $\boldsymbol{d}$ and perform a line search along $\boldsymbol{d}$ by changing the step size η.
To account for how the different weights in $\boldsymbol{w}$ have varying degrees of influence on the response of an input, which Ref. [25] refer to as sensitivity, one can "normalize" the perturbation with respect to the sensitivity of that weight. Because the mutation direction $\boldsymbol{d}$ is sampled uniformly the divergence can become dominated by the most "sensitive" weights. This could lead to the problem of only exploring the behaviours characterized by a much smaller set of weights, slowing or potentially even limiting the search. If the network is differentiable with respect to weights we can approximate how the output changes in a region around $\boldsymbol{w}$ by doing a first order taylor expansion

$$y_{ik}(\boldsymbol{x}_i, \boldsymbol{\Delta w}; \boldsymbol{w}) \approx \Phi(\boldsymbol{x}_i; \boldsymbol{w})_k + \boldsymbol{\Delta w}\nabla_{\boldsymbol{w}}\Phi(\boldsymbol{x}_i; \boldsymbol{w})_k \tag{2.7}$$

and investigate the sensitivity of the weights $\boldsymbol{w}$ at $\boldsymbol{x}_i$ for output k. Here the scalar product $\boldsymbol{\Delta w}\nabla_{\boldsymbol{w}}\Phi(\boldsymbol{x}_i; \boldsymbol{w})_k$ describes how much the networks kth output changes when we apply perturbation $\boldsymbol{\Delta w}$.
An efficient way to "normalize" $\boldsymbol{\Delta w}$ is coefficient-wise division of each parameters sensitivity s_j

$$\boldsymbol{\Delta w}_{normalized} = \{\frac{\Delta w_j}{s_j}\}_{j=1}^{dim(\boldsymbol{w})} \tag{2.8}$$

where the sensitivity vector $\boldsymbol{s}$ is approximated as in Ref. [25] by

$$\boldsymbol{s} = \sqrt{\sum_{k=1}^{m} \Big[\sum_{i=1}^{N} \nabla_{\boldsymbol{w}} \Phi(\boldsymbol{x}_i; \boldsymbol{w})_k \Big]^2} \tag{2.9}$$

which is referred to as *summed gradient* sensitivity. It should be clarified that in equation 2.9 the sums, square and root operations are carried out in a coefficient wise manner of the gradient $\nabla_{\boldsymbol{w}} \Phi(\boldsymbol{x}_i; \boldsymbol{w})_k$.

2.2.4 Selection operators

A selection operator applies selective pressure to a population, such that more fit individuals propagate through the generations. Too high selective pressure and the algorithm might converge prematurely. It is sometimes referred to as *take over time* [13, p. 135], the time it takes for a solution to dominate the population. Depending on the exploration needed for a given problem, the selective pressure is of utmost importance for the performance of the algorithm.
There are many types of developed selection operators such as *random selection*, *tournament selection* and *proportional selection* [13, p. 135-139]. The operator used in this project for pairing mates is tournament selection. Tournament selection is where you take a number, referred to as the *tournament size*, of random individuals and order them by fitness. Selecting the best with a probability p_{tour} and if the individual is not selected the second best is selected with p_{tour}. These "trails" repeat until only the least fit is remaining which is then selected [12, 13]. Typical values of p_{tour} are between $0.5 - 0.9$ [12].

2.2.5 Replacement operators

The replacement operator determines how the population is maintained. That is how offspring and parents get the opportunity to persist across generations. It is common to replace all parents from one generation with an equal amount of offspring, meaning there is no overlap of parents and offspring. To name a few replacement operators we have *replace worst*, *kill tournament*, *parent- offspring competition* [13, p. 158]. A replacement operator replacing an individual as a function of its fitness will apply additional selective pressure. These surviving individuals can in that case compete across generations which increases convergence towards that individual. An example of this is the so called *elitism operator*, where a number of the best individuals are always carried over to the next generation [12, 13].
We let an individual be killed off with a linearly increasing probability in proportion to its rank in the population. The probability is 0 for a number of the best, see Appendix B, and then increases to 1 for the lowest ranking individual.
In the case of offspring not replacing parents directly, i.e crossover and replacement operating independently, the result is a dynamically sized population which we have to guarantee a finite steady state for in order to avoid total extinction or exploding population sizes. One reason for using dynamic populations is that together with *dynamic niching* (introduced later) we can more efficiently explore the parameter space by controlling the local population density, by having a smaller total population size. Many different models have been developed with seemingly good results,

with the ability to explore multiple solutions by populating the optimization problems pareto frontier[1] in multi-objective (MO) optimization problems [31, 32]. For example, an optimization problem consisting of minimizing energy and maximizing forward velocity might have two or more equal optima where one uses less energy at the cost of slower movement.

2.2.6 Speciation by dynamic niche clustering

Speciation or *niching* is the concept of dividing a population of individuals into groups that are similar on the genetic level. Biologically an individual is often only able to mate with a member from the same species. This might be because of chromosomal structure, or generally mismatching genomes so that genes which encodes vital functionality might get lost during crossover. Species also often occupy their own ecological niche where they compete for food and reproduction opportunities, which corresponds to using the replacement, selection and crossover operator on a subset of the population. Reasons for modeling this include searching for multiple solutions (usually the case in multimodial optimization problems), maintaining diversity, and allowing new but unfit solutions to survive if they are different, in hope that they will lead to a better optimum.
The primary method used for static networks in our implementation is a form of *dynamic niche clustering*. Our population of networks is divided by assigning each member to different niches based on their genetic similarity. The method can be summarized as in Ref. [13, p. 168] by the following steps

1. Assign each member of the initial population to a niche associated with a merge radius and maximum niche radius. The midpoint, $\boldsymbol{x}_\mu$, of each niche will initially be $\boldsymbol{x}_i$, the genome of individual i.
2. Add new members of the current generation to any niche where $|\boldsymbol{x}_\mu - \boldsymbol{x}_i| < r_{max}$. If no niche is found for a new individual, create a new niche if the number of niches is below a maximum number of niches, add to the closest niche otherwise.

$$\boldsymbol{x}_\mu = \boldsymbol{x}_\mu + \frac{\sum_{i=1}^{n_\mu} (\boldsymbol{x}_i - \boldsymbol{x}_\mu) \cdot f(\boldsymbol{x}_i)}{\sum_{i=1}^{n_\mu} f(\boldsymbol{x}_i)} \tag{2.10}$$

   where $\boldsymbol{x}_\mu$ is the center of niche μ, n_μ the number of individuals in niche μ and $\boldsymbol{x}_i$ is the encoded weights and parameters in the network of individual i with fitness $f(\boldsymbol{x}_i)$.
3. Merge any two niches whose midpoints that are closer then r_{merge} in euclidean distance.
4. If the number of members of a niche exceeds a certain threshold, try splitting the niche randomly into two parts with decreased merge radius.
5. Increment the merge and max radius to slowly merge the different niches. Repeat from step 2.

[1] A pareto front is where a measure of performance cannot get better without making another measure worse. This is sometimes referred to as *pareto optimal tradeoff surface* and exploring it can be difficult because of valleys or hills in the fitness landscape.

Figure 2.4 illustrates the population divided into their own niches, which allows them to search locally even if there is a globally more fit solution that would otherwise dominate them.

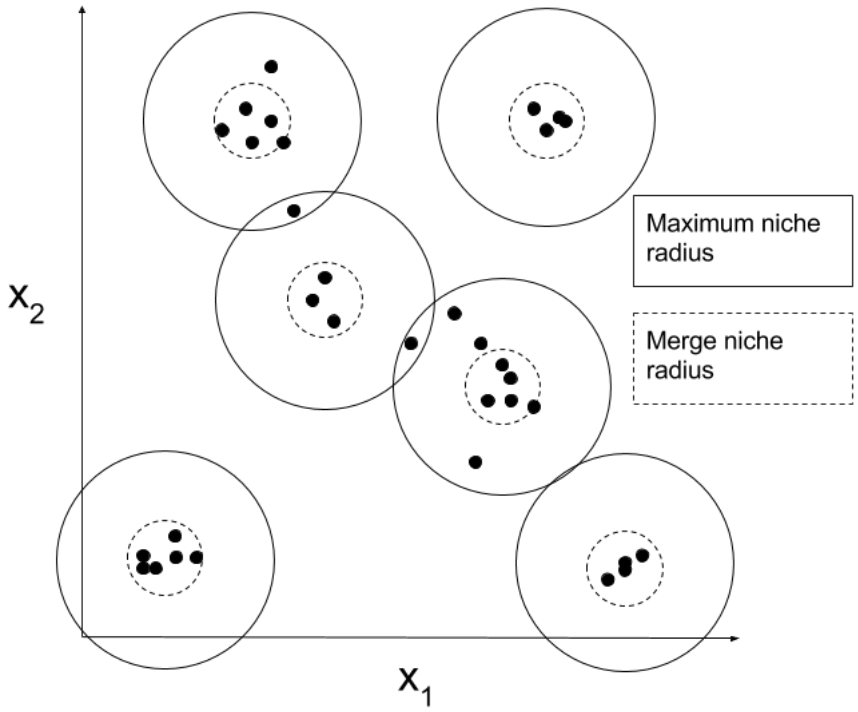

Figure 2.4: A population of candidate solutions (the black dots) with parameters $\boldsymbol{x}_i = (x_{1i}, x_{2i})$ divided into niches. A new individual belongs to a niche if it is inside a solid circle, note that a individual can thus belong to more then one niche (two such cases where in the figure where circles overlap). If the dashed circles intersect the corresponding niches are merged and a new maximum niche radius is found so that all individuals are included.

This technique allows us to search for non-dominating solutions as the other operators are now applied to each niche allowing less fit but different solutions to evolve. This also allows for continued search by applying crossover between niches even though all local sub-populations might have converged.

2.3 Reinforcement Learning

2.3.1 Common Reinforcement Learning Concepts

Reinforcement learning tackles a generalized variant of the so called *Markov Decision Process* (MDP) problem. An MDP consists of a finite state space S, a finite action space A, transition probabilities $P_a(\mathbf{s}_t, \mathbf{s}_{t'})$, rewards $r_a(\mathbf{s}_t, \mathbf{s}_{t'})$, and a discount factor $\gamma \in [0, 1]$. An agent in this world can at each discrete time step t chose an action $\mathbf{a}_t \in A$ which with probability $P_a(\mathbf{s}_t, \mathbf{s}_{t'})$ will transition the state into $\mathbf{s}_{t'}$ at the next time step. The agent will then receive feedback on its choice of action in the form

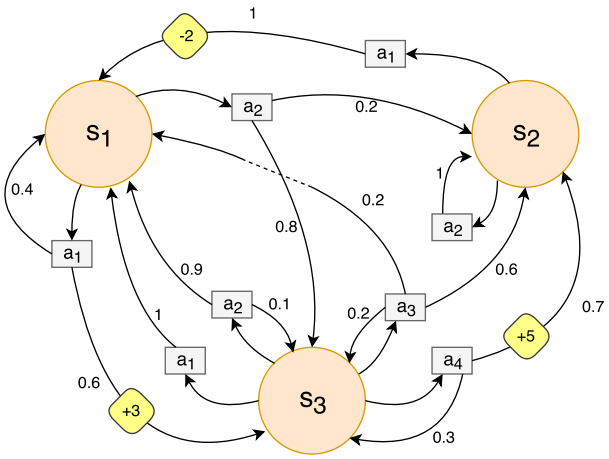

Figure 2.5: A simple Markov Decision Process featuring 3 states, a few possible actions in each state, transition probabilities, and sparse rewards in yellow.

of a reward $r_a(\mathbf{s}_t, \mathbf{s}_{t''})$ [2]. Unless some terminating conditions are met, the process will then repeat, and the agent will observe the new state and select a new action. See Figure 2.5.
The goal is to find a policy function $\pi(\mathbf{s}_t)$ which specifies the optimal action $\mathbf{a}_t$ to take, given a state $\mathbf{s}_t$, in order to maximize the cumulative (possibly discounted) reward given by

$$\sum_{t=0}^{T} \gamma^t r_{\mathbf{a}_t}(\mathbf{s}_t, \mathbf{s}_{t+1})$$

where γ is the discount factor determining how heavily to weigh short term rewards over future ones. This quantity is often referred to as the return. An MDP problem can be efficiently and optimally solved using *dynamic programming* [33] or *linear programming* [34] as long as the transition and reward functions are known and action and state spaces are finite.
But what happens if one does not have perfect information, if the action or state spaces are infinite, or the functions P_a and $r_a(\mathbf{s}_t, \mathbf{s}_{t'})$ are unknown? This is the more general problem reinforcement learning intends to solve. It is still helpful to view the problem as a series of states $\mathbf{s}_t$, actions $\mathbf{a}_t$, and rewards r_t, but without assuming anything about the structure of transition probabilities or reward functions, see Figure 2.6. The goal remains to find a policy function $\pi(\mathbf{s}_t)$ that maximizes the expected return, starting from an initial state $\mathbf{s}_0$ determined by some initial state distribution. This policy function may take the more general form of a distribution, in which case we will denote its probability density function by $\pi(\mathbf{a}_t|\mathbf{s}_t)$. The output is then the probability density of action $\mathbf{a}_t$ given the state $\mathbf{s}_t$. During runtime when the policy needs to select an action, a sample is produced from this action distribution.
To help facilitate further explanations the following common reinforcement learning concepts are introduced:

[2] Note that the state $\mathbf{s}_{t''}$ is the state that was actually transitioned into which depends on the P_a distribution

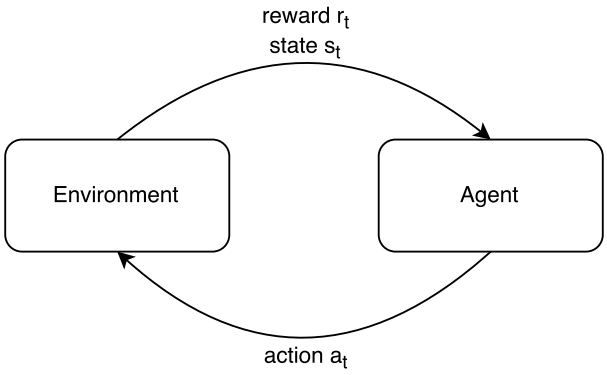

Figure 2.6: The basic reinforcement learning cycle, consisting of state-action-reward tuples.

- **The Value Function** $V_\pi(\mathbf{s}_t)$: denotes the expected discounted return continuing from a state $\mathbf{s}_t$ if future actions are sampled from policy π;

$$V_\pi(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t, s_{t+1}, \ldots \sim \pi} \left[\sum_{l=0}^{\infty} \gamma^t r_{t+l} \right].$$

  By $V^\star(\mathbf{s}_t)$ we denote the optimal (maximum) value of a state, achieved if one was to follow an optimal policy maximizing the return starting from this state.

- **The State-Action Value Function** $Q_\pi(\mathbf{s}_t, \mathbf{a}_t)$: denotes the expected return in a state $\mathbf{s}_t$ if the chosen action at time t is $\mathbf{a}_t$ and the remaining future actions are sampled from the policy π;

$$Q_\pi(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}_{s_{t+1}, a_{t+1} \ldots | \mathbf{a}_t} \left[\sum_{l=0}^{\infty} \gamma^t r_{t+l} \right].$$

  This function is closely linked to the Value Function in the sense that if action $\mathbf{a}_t$ is also sampled from the policy π then the following holds;

$$Q_\pi(\mathbf{s}_t, \mathbf{a}_t)|_{\mathbf{a}_t \sim \pi} = V_\pi(\mathbf{s}_t).$$

  Note that if these Q-values are known for all $\mathbf{s}_t$ and $\mathbf{a}_t$ for an optimal policy $\pi^\star$, then the optimal action is to always chose the action that maximizes Q

$$\pi^\star(\mathbf{s}_t) = \underset{\mathbf{a}_t}{\operatorname{argmax}} \left(Q_{\pi^\star}(\mathbf{s}_t, \mathbf{a}_t) \right).$$

  Some traditional reinforcement learning algorithms like SARSA [35] and Q-learning [36] exploit this fact by keeping track of Q-value estimates for all possible state-action pairs. Through some forced exploration these tables are iteratively updated with new values, and the algorithms converge towards an optimal policy. This procedure is unfeasible when dealing with continuous action- and state-spaces as these tables would not be finite.

- **The Advantage Function** $A_\pi(\mathbf{s}_t, \mathbf{a}_t)$: denotes the expected change in return, or advantage, of taking action $\mathbf{a}_t$ instead of following a policy's default behaviour. Just as with $Q_\pi(\mathbf{s}_t, \mathbf{a}_t)$ action $\mathbf{a}_t$ is taken at state $\mathbf{s}_t$ and thereafter actions are decided by the policy π. The difference being that this Q-value is baselined with the expected default behaviour of following the policy; given by $V_\pi(\mathbf{s}_t)$;

  $$A_\pi(\mathbf{s}_t, \mathbf{a}_t) = Q_\pi(\mathbf{s}_t, \mathbf{a}_t) - V_\pi(\mathbf{s}_t).$$

  It follows that a positive advantage value corresponds to actions yielding higher expected returns than the default policy behaviour. Similarly, a negative advantage value corresponds to actions yielding lower expected returns than the default policy behaviour.

2.3.2 Policy Gradient Methods

The idea behind policy gradient methods [37, 38] is quite simple. First we assume our policy $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ is parameterized by some known parameter vector $\boldsymbol{\theta}$. By changing these parameters one can alter the behaviour of the policy. Ideally we would like to know in which direction in parameter space to tweak $\boldsymbol{\theta}$ in order to increase the expected return. This can be determined by calculating the gradient with respect to $\boldsymbol{\theta}$. For a general function $f(x)$ where x is a randomly distributed variable, the following holds

$$\begin{aligned}\nabla_\theta \mathbb{E}_x\left[f(x)\right] = \nabla_\theta \int p(x|\boldsymbol{\theta}) f(x) dx = \int \nabla_\theta p(x|\boldsymbol{\theta}) f(x) dx = \\ = \int \frac{p(x|\boldsymbol{\theta})}{p(x|\boldsymbol{\theta})} \nabla_\theta p(x|\boldsymbol{\theta}) f(x) dx = \int p(x|\boldsymbol{\theta}) \nabla_\theta \log(p(x|\boldsymbol{\theta})) f(x) dx = \\ = \mathbb{E}_x\left[\nabla_\theta \log(p(x|\boldsymbol{\theta})) f(x)\right]. \quad (2.11)\end{aligned}$$

By applying similar computations to the notion of policy gradients (without a discount factor for simplicity) one arrives at

$$g \equiv \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} r_t\right] \approx \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} r_t \nabla_\theta \log(p(\tau|\boldsymbol{\theta}))\right] \quad (2.12)$$

where $\tau \sim \pi_\theta$ is the entire coherent sequences of events, referred to as a trajectory,

$$\tau = \{\mathbf{s}_0, \mathbf{a}_0, r_0, \mathbf{s}_1, ..., \mathbf{s}_{T-1}, \mathbf{a}_{T-1}, r_{T-1}, \mathbf{s}_T\}$$

with actions sampled from the policy π_θ and states and rewards determined by the environment. Note that for this computation to hold we implicitly make the assumption that the reward series is independent of the trajectory which is clearly not true. Nevertheless it leads to a highly functional update rule. In Sections 2.3.3 & 3.2.2 other objectives besides the return are maximized, but they still maintain the problem of not actually being independent of τ.
The gradient of the logarithmic probability of the trajectory can be calculated as follows

$$\nabla_\theta \log(p(\tau|\theta)) = \nabla_\theta \log\left(p(\mathbf{s}_0) \cdot \prod_{t=1}^{T-1} \pi(\mathbf{a}_t|\mathbf{s}_t, \theta) \cdot p(\mathbf{s}_{t+1}, r_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\right) =$$

$$\nabla_\theta \left(\log(p(\mathbf{s}_0)) + \sum_{t=0}^{T-1} \log\left(\pi(\mathbf{a}_t|\mathbf{s}_t,\theta)\right) + \sum_{t=0}^{T-1} \log\left(p(\mathbf{s}_{t+1}, r_{t+1}|\mathbf{s}_t,\mathbf{a}_t)\right) \right) = \\ = \sum_{t=0}^{T-1} \nabla_\theta \log(\pi(\mathbf{a}_t|\mathbf{s}_t,\theta))$$

where many terms disappeared as they were independent of θ. Plugging this back into Equation (2.12) gives:

$$g = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t'=0}^{T-1} r_{t'} \sum_{t=0}^{T-1} \nabla_\theta \log(\pi(\mathbf{a}_t|\mathbf{s}_t,\theta)) \right]. \tag{2.13}$$

In order to deal with the expectation value we take Monte Carlo estimates, sampling all values along a simulated trajectory, in order to produce an estimator $\hat{g}$ of the gradient g. In practice the trajectories can cover thousands of timesteps and it is more efficient to calculate each timestep-dependent component of the gradient by itself and perform the updates in smaller pieces;

$$\hat{g} = \sum_{t=0}^{T} \hat{g}_t \quad , \quad \hat{g}_t = \sum_{t'=0}^{T-1} r_{t'} \nabla_\theta \log(\pi(\mathbf{a}_t|\mathbf{s}_t,\theta)). \tag{2.14}$$

Assuming that the reinforcement learning environment can be reset and run at our will, we can use it to generate many different trajectories for any given policy π_θ. Even when using the same π_θ these trajectories will differ from one another due to:

- Possible differences in initial states $\mathbf{s}_0$.
- The policy being stochastic. The chosen action at each timestep is a random variable sampled from a distribution.
- The environment might not be deterministic and can transition into different states from equivalent state-action pairs.

The $(r_t, \mathbf{a}_t, \mathbf{s}_t)$ data points are shuffled over time and over multiple trajectories in order to reduce correlation before calculating the gradient estimates $\hat{g}_t$. Parameter updates are then conducted according to the chosen parameter updating scheme in smaller blocks of data known as mini-batches.

To summarize; the resulting system thus alternates between two procedures. The first is simulating a batch of trajectories under the current policy. The second is to optimize policy parameters to increase the odds of performing the sampled actions relative to the return yielded by their trajectory. This is the vanilla policy gradient method which will be improved upon in the following sections.

2.3.3 Advantage Estimation

Equations (2.13) and (2.14) weigh every action taken during a trajectory evenly, according to the trajectories final return. One simple improvement is to weigh an action only relative to the rewards received after that action was taken, as the action clearly can not have an effect on rewards received before it was performed. This leads to a gradient estimate of

$$\hat{g}_t = \sum_{t'=t}^{T-1} r_{t'} \nabla_\theta \log(\pi(\mathbf{a}_t|\mathbf{s}_t,\theta)). \tag{2.15}$$

Better yet would be to encourage and discourage actions depending on if they are better or worse than what the policy is already doing. The measure for this is the advantage Function A_π from sec 2.3.1,

$$\hat{g}_t = A_{\pi,t} \nabla_\theta \log(\pi(\mathbf{a}_t|\mathbf{s}_t, \theta)). \tag{2.16}$$

With this gradient estimator, actions with negative advantage will move in the opposite of the probability gradient direction, reducing the odds of them being selected in the future by the policy. However, the components of the advantage function, the value-function V_π and the state-action value function Q_π, also need to be estimated. The estimate of the state-action value function is typically taken to be the empirical reward series $\sum_{t'=t}^{T} \gamma^{t'-t} r_t$ along the corresponding trajectory being investigated. The value-function is typically handled as a supervised learning problem where another neural net is trained on the discounted Monte Carlo sampled reward series, along each trajectory, see Section 2.3.5. The result being an actual function which can estimate the return from a given state under the policy. As a result of training data being centered around frequently followed trajectories, exploration which leads to states far from the norm might be improperly valued. Thus the value function estimator needs to converge towards accurate estimates before the policy can converge towards an optimum. This relationship between value function and policy can be a sensitive issue when training.
In practice [10, 17] it turns out that normalizing the advantage estimates to zero mean and unit variance has beneficial properties for the learning process. This does mean that half of all actions in a batch of trajectories receive a positive advantage estimate and are encouraged during training, with the other lower scoring half being discouraged. This does go against some of the logical reasoning above, but our experiments also support this modification of the theory.

2.3.4 Neural Networks as Policy Functions

As previously stated a policy function π is responsible for deciding what action to take in a given state. For the learning environments regarded in this work the state and action spaces are multidimensional, and the policy needs to be parameterized and differentiable with respect to those parameters. A stochastic policy is also desirable in order to force greater exploration of the policy space. For these reasons all polices in this work are represented with the use of neural networks, as has become the norm in reinforcement learning of many sorts in the last few years [14, 16, 39]. The vector of outputs from the network is in turn used to parameterize a multidimensional Gaussian distribution, with diagonal covariance matrix (the dimensions are uncorrelated), see Figure 2.7. Note that the resulting probability density function is still differentiable with respect to every weight in the network.
With such a multivariate Gaussian probability density function

$$\begin{aligned}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) &= P(a_{t,1}, a_{t,2}, ..., a_{t,n}|\mathbf{s}_t) = \\ &= \frac{1}{(2\pi)^{n/2}\sigma_1\sigma_2 \cdot ... \cdot \sigma_n} \exp\left(-\frac{(a_{t,1}-\mu_1)^2}{2\sigma_1^2} - \frac{(a_{t,2}-\mu_2)^2}{2\sigma_2^2} - ... - \frac{(a_{t,n}-\mu_n)^2}{2\sigma_n^2}\right)\end{aligned} \tag{2.17}$$

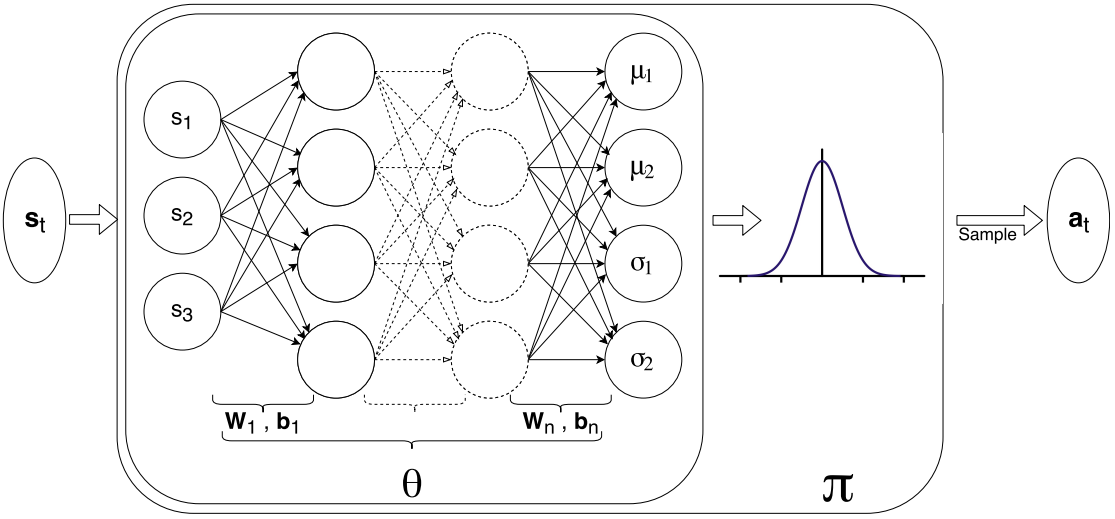

Figure 2.7: A mapping of a neural network to a stochastic policy function π, mapping from a 3 dimensional state space to a 2 dimensional action space. All weights and biases are defined in the parameter vector θ.

the local parameter derivatives become (see appendix A for calculations)

$$\frac{\partial}{\partial \mu_i}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \cdot \left(\frac{a_{t,i} - \mu_i}{\sigma_i^2}\right)$$

$$\frac{\partial}{\partial \sigma_i}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \cdot \frac{1}{\sigma_i}\left(\left(\frac{a_{t,i} - \mu_i}{\sigma_i}\right)^2 - 1\right).$$

These derivatives show us how to change the parameters to increase or decrease the likelihood of a sample. For example notice how the sign of the sigma derivative depends on whether or not the sampled action is within one standard deviation of the mean. They can be used with normal backpropagation [40] during the training procedure of the neural network.
Alternatively, one can set up the NN to only output the μ values while controlling the variances separately, see Figure 2.8. Even though σ is not connected to the network one can still backpropagate the error gradient to it, updating it like any other parameter. A third alternative is to update σ 'manually' over time. In such a system the exploration-exploitation trade off (determined by σ) can be managed in a more controlled manner, at the cost of requiring additional parameters to fine tune. For example in Ref. [17] the standard deviation vector is equivalent over all dimensions and decays exponentially from a large initial value down to a lower threshold over the course of training. However, with such a setup of equivalent σ_i values over dimensions i the policy loses the capacity to model different levels of certainty over the different action space dimensions. Arguably an acceptable price to be paid for additional control over the training procedure.
Regardless of the choice of training scheme, the variance vector will be set to zero at runtime[3]. The policy will then exhibit deterministic behaviour, as additional

[3]this is when all training has concluded and one wishes to asses the final performance of the policy

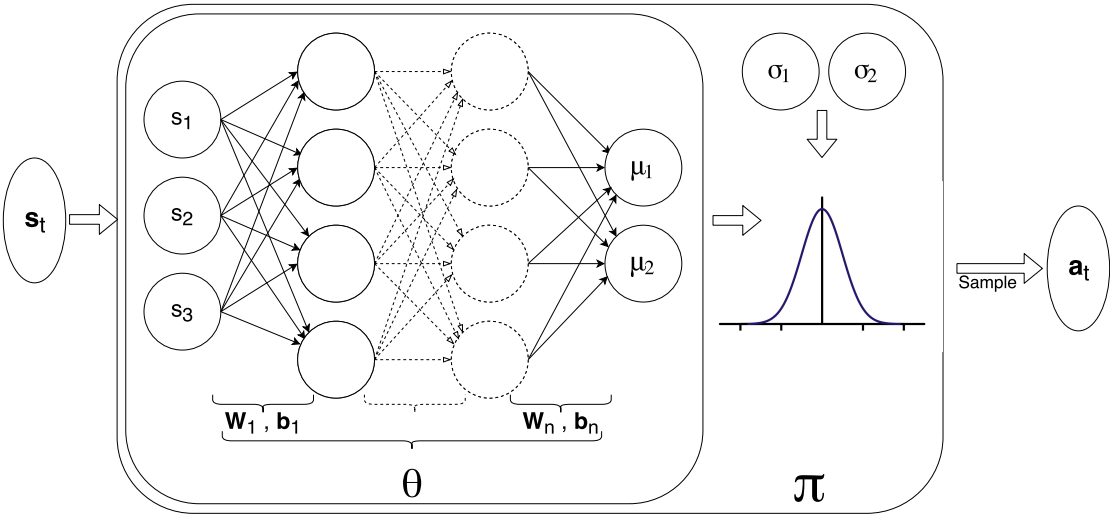

Figure 2.8: A mapping of a neural network to a stochastic policy function π, with σ independent of the neural network.

exploration is no longer desired. One merely wants to examine the already learned behaviour.

2.3.5 Neural Networks as Value Functions

To model the value function a second neural net is used. For simplicity the same architecture is used as for the policy network with the difference being that it only has a single scalar output, regardless of the dimensionality of the action space, see Figure 2.9. This output is an estimate of the return of a trajectory continuing from the state fed to the network as input. The trajectories generated for the policy network are at the same time used to calculate empirical returns from different states. This is then used to train the value network in mini-batches in conjunction with the training procedure of the policy network.
It is worth noting that the value and policy network relationship can be viewed as an actor-critic system [41]. The value network provides feedback on the actions chosen by the policy network, guiding its learning.

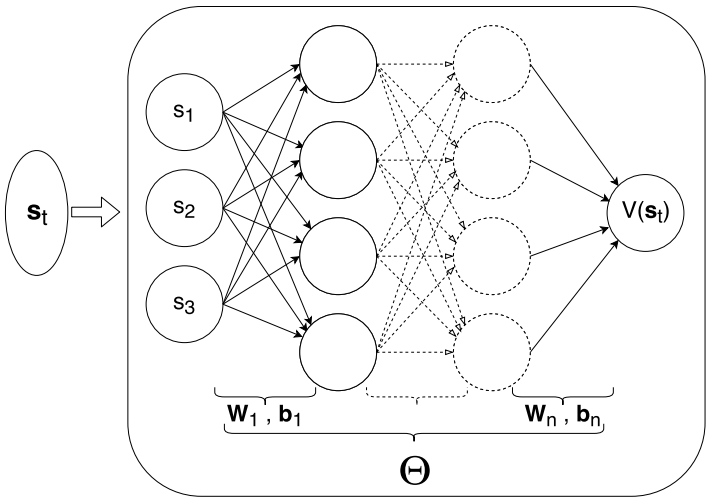

Figure 2.9: A neural network as a value function. Weight and biases are defined in parameter vector Θ. The value function outputs a scalar value approximating the return when starting in state $\mathbf{s_t}$ and following the policy.

3
Machine Learning and Locomotion

Both the evolutionary and reinforcement learning approaches have previously produced viable locomotion for their agents with some robustness, i.e. tolerance to disturbances in movement and ability to deal with new environments. Here ideas and methods from this previous work are presented, which we make further use of in our own application.

3.1 Evolutionary Algorithms for Locomotion

Multiple models have previously been proposed to solve this locomotion challenge by making use of evolutionary algorithms, such as in Ref. [42] where they use "evolving neural nets" that can evolve its architecture while also injecting "predefined nets" with walking abilities into the population to improve solution performance. In Ref. [7] they take on the 3D humanoid locomotion challenge by increasingly allowing their neural nets to grow. A popular method is to use *continuous time recurrent neural networks* such as in [8, 9], where they use fully connected but "static" neural networks as controllers. These methods typically involves taking parameters for a neural network and encoding them into a genome. The genome represents all chosen parameters to be subject for evolution such as the links between neurons (network architecture), their synaptic strength or weights, even activation functions for any particular neuron in the network as in [43]. Then for each generation a network has to be reconstructed from the genome in order to evaluate its fitness, i.e how useful locomotive abilities it can induce in a body.
There are also more traditional controllers, see [44], which use genetic algorithms[1] (GAs) to find an optimal set of parameters for their regulators. Using evolutionary algorithms to train neural networks however comes with its own set of disadvantages and advantages. Evolutionary algorithms are inherently parallel and can be scaled well by adding more hardware. Taking the scalability into account and using simple mutation and recombination operator such as in *evolutionary strategies* (ES)[2], recent studies claim that they are competitive with reinforcement learning algorithms in terms of performance [45]. In [46] they compare ES and a standard GA with *novelty search* [47], where the GA outperforms ES in training deep neural networks for hard locomotion problems such as making a 3D humanoid walk and get past obstacles. The article [46] remarks that older techniques developed in the neuro-evolution

[1] Genetic algorithms is a subclass of evolutionary algorithms.
[2] Evolutionary strategies is also a subclass of EA. Consists of producing slightly mutated offspring from one individual and recombining the entire population in a weighted manner.

community can still provide very good results.
Evolutionary algorithms are also widely used in evolutionary robotics; the study of evolving locomotion behaviour or gaits in different mechanical bodies [48]. Transferring an agent from a simulated environment to a real robot also has the added difficulty of *noise and reality gap* [49], showing that evolutionary algorithms can find robust neural network controllers.
One of the problems with evolutionary algorithms is that the hyper-parameters[3] required are sometimes very problem dependent. The objective function also plays an important role for the evolution of a desired controller. One will often find agents getting stuck in poorly performing local optima with simpler objective functions because the landscape might favor slow or stiff movement due to the increased stability. This problem of premature convergence is discussed in Section 4.1. Many find that maintaining diversity is paramount for solving more difficult optimization problems [50]. One way to achieve this is by dividing the population into species or niches [26].

3.2 Reinforcement Learning for Locomotion

Within the field of reinforcement learning control tasks of different sorts are nothing new. On a fundamental level reinforcement learning always tries to teach an agent to select appropriate actions, in other words output some control signal to its environment. The main issue with our setting is that many of the traditional RL algorithms are designed to operate on discrete action and/or state spaces. These algorithms are incompatible with our work as the state space will consist of a body's sensory outputs, such as joint positions, joint velocities, body velocity, accelerometers, touch sensors, range finders, etc. The action space will be inputs to motors/muscle actuators. Both these action- and state-spaces are multidimensional continuous spaces, although they are often restricted to certain intervals.
To illustrate the difference; our locomotion problem can be discretized naively by binning the continuous action space to only output binary controls of +1,-1. This still leads to exponential growth in the output space of size 2^n where n is the number of actuators, while obviously only offering a very coarse controller. For these reasons it is desirable to remain in the continuous space and use algorithms tailored for such scenarios.
Another reoccurring challenge when developing RL algorithms is the need of a handcrafted fitness measure or reward function in order to avoid problems with robustness, and to steer the agent towards the designer's desired results. In [10] Google's Deepmind team makes use of reinforcement learning with an abnormally simple reward function in conjunction with a rich and diverse environment. They claim this approach leads to the emergence of richer and more robust behaviour of the agents. Thus, instead of having to be explicitly punished by modifications to the reward function, many undesired behaviours are avoided due to their lack of generalizing to alterations in environment.

[3] The control parameters for the algorithm that governs for example exploration such as mutation rate.

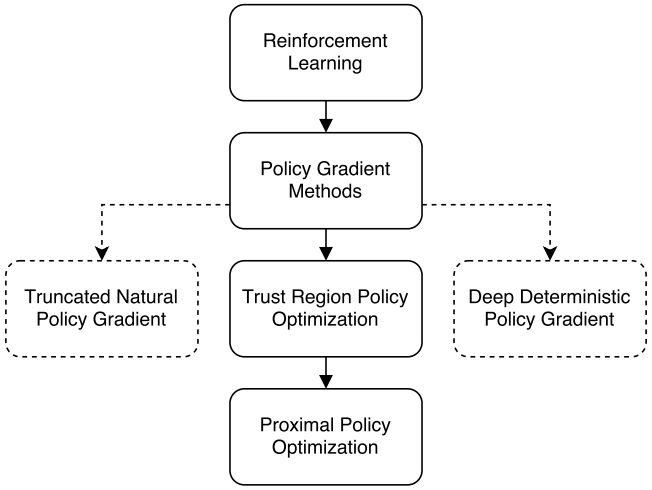

Figure 3.1: Hierarchical relationships between the main reinforcement learning algorithms discussed. The ones with dotted lines are not of further importance in this work.

In Ref. [18] a number of RL algorithms are benchmarked against each other in a variety of continuous control tasks. Truncated Natural Policy Gradient (TNPG), Trust Region Policy Optimization (TRPO), and Deep Deterministic Policy Gradient (DDPG) outperform the others. They are extensions of the more fundamental Policy Gradient method presented in Section 2.3.2, see the inheritance chart in Figure 3.1. Although they all have their own strengths and weaknesses, TRPO [16] and its variation Proximal Policy Optimization (PPO) [17] fit the purpose of this work best. Both have showed promising results in solving locomotion tasks similar to ours [18]. TNPG and DDPG will not be of further interest in this work, while TRPO and PPO are presented below.

3.2.1 TRPO - Trust Region Policy Optimization

In Ref. [16] a surrogate objective function is presented. By maximizing this surrogate objective instead of the expected return, monotone improvement of the return is still achieved. The justifying theory leads to an estimator that is hard to utilize, but can be approximated as follows

$$\underset{\theta}{\text{maximize}}\ \hat{\mathbb{E}}\left[\frac{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}{\pi_{\theta old}(\mathbf{a}_t|\mathbf{s}_t)}\hat{A}_t\right]$$

$$\text{subject to}\ \hat{\mathbb{E}}\left[KL[\pi_{\theta old}(\cdot|\mathbf{s}_t), \pi_\theta(\cdot|\mathbf{s}_t)]\right] \leq \delta$$

A constraint has been added to keep the policy updates from being detrimentally large at each iteration. It utilizes the Kullback-Leibler divergence, a measure of how well data from one distribution can be explained by another. It can intuitively be thought of as a distance measurement between two distributions, although this is not strictly correct. By limiting the KL-divergence changes in the policy during a single iteration of the algorithm are not allowed to change the policy too much at once, keeping it within the so called *trust region* of the policy from the previous

iteration. TRPO has proven to be well adapt at handling continuous control tasks, but is usually outperformed by its younger brother PPO.

3.2.2 PPO - Proximal Policy Optimization

Instead of optimizing with a constraint as in TRPO the authors of Ref. [17] use a clipping method to hinder too large policy changes. The optimization problem then becomes

$$\underset{\theta}{\text{maximize}}\ \hat{\mathbb{E}}\left[min\left(\omega_t(\theta)\hat{A}_t, clip\left(\omega_t(\theta), 1+\epsilon, 1-\epsilon\right)\hat{A}_t\right)\right]$$

where ϵ is a parameter controlling the bounds of the clipping function, and ω_t is the ratio of the probabilities of the action at timestep t being taking under the new versus old policy:

$$\omega_t(\theta) = \frac{\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}{\pi_{\theta old}(\mathbf{a}_t|\mathbf{s}_t)} \quad \text{and} \quad clip(x,u,l) = \begin{cases} l, & x < l \\ x, & l < x < u \\ u, & x > u \end{cases} .$$

This deceptively simple clipping function achieves a similar trust region effect as the TRPO constraint and yields better results in practice. Calculating the derivatives one can see that, when fed a sample, if the updated policy acts outside of the trust region[4] the derivatives are zeroed if they were to increase the objective but unaffected if they decrease it. Thus there is no incentive for the policy to wander out of the trust region as this can only lead to a lower scoring objective value.

[4]This is when ω is out of bounds and the clipping function takes effect. The authors recommend $\epsilon \approx 0.2$

4

Method

In this chapter we detail the application of the theory to the problem of optimizing locomotion behaviour on a practical pseudo-code level. We also look at a hybrid method making use of both EA and RL for the same purpose, as well as detailing how everything was implemented on a programming level.

4.1 The Evolutionary Algorithm Method

The operators used for the evolutionary algorithm are the ones presented in chapter 2 with more details. Pseudo code can be found in Algorithm 1 with parameters in Appendix B for each problem.

Result: A population of solutions
generation = 0;
Population = 128 Xavier initialized neural networks.;
Niches = make niche for each individual in Population;
while *generation* < *MaxGenerations* **do**
    Evaluate Population with objective function;
    Update Niches;
    Sort each niche population by fitness and the total Population;
    Apply **replacement** operator to each niche;
    Apply **directional crossover** for each niche using **tournament selection**;
    Apply **directional crossover** for each niche with a number of random niches using **tournament selection**;
    Apply **weighted multiparent crossover** on entire population using **tournament selection**;
    Apply **mutation** operator if rank in niche < elitism count;
    Add new Individuals from crossover to Niches and Population;
end

Algorithm 1: The EA implementation for training locomotion producing neural networks. Note that this allows for each niche to produce their own unique gait since they are separated from evolutionary pressure from other niches. Also note that the best individuals in each niche are not modified by the mutation operator so that we do not destroy them.

4.2 The Reinforcement Learning Method

The PG and PPO reinforcement learning algorithms presented in chapter 2 were implemented as discussed along with some experimentation regarding how they should interface to the locomotion environment. Xavier initialization [51] was used to avoid over-saturating neurons in the initial network state. The policy outputs are truncated to match the control range of $[-1, 1]$. Whenever the environment resets, the initial body position is altered with some uniformly distributed noise to offset starting joint positions and velocities. A test trajectory is aborted early if the body reaches a fatal state, for example a biped falling over. Advantage value estimation was done as detailed in Section 2.3.3. A secondary network is used to approximate the state-value function. The mapping of a neural network to a policy function was also experimented with, according to the three different ways of handling standard deviation vectors from Section 2.3.4. Pseudo-code for the algorithm;

```
while i < max_iter do
    while traj < batch_size_traj do
        while t < max_t do
            if check_terminating_conditions() then
                continue;
            else
                action_t ← policy.sample(state);
                state_t, reward_t ← env.step(action);
            end
        end
        V ← valuenet.makeValuePredictions(states);
        Adv ← estimateAdvantangeValues(V, rewards);
    end
    Shuffle trajectory data
    Train valuenet parameters Θ with empirical errors in minibatches
    Train policynet parameters θ by maximizing surrogate PPO objective, a
     function of Adv, θ, and θ_old.
    θ_old ← θ
end
```

Algorithm 2: The Proximal Policy Optimization procedure.

4.3 The Hybrid Method

In order to take advantage of the strengths of both RL and EA methods we also tested a hybrid optimization method. The main idea is to first do a coarse optimization through EA leading to a population of different local optima. The better performing candidates can then be further refined through reinforcement learning. Why this seemed a promising approach we discuss in Section 6.4.

The networks trained through EA can be directly substituted into a policy network estimating mean control values. The standard deviations are then handled separately as in figure 2.8. When starting the RL procedure, the separate value-network will be completely untrained on its new policy counterpart and only estimate nonsense. Policy updates calculated from such an uncalibrated value network can be very detrimental and quickly destroy any prelearned behaviour. For this reason a 'burn in' of some number of generations is performed. During this time, generated trajectories are only used for training the value network and not to calculate updates to the actual policy. The variance is also cranked up during these trajectories; encouraging a wider range of states to be reached and offering some compensation for missed training experience.
The resulting agent is hopefully equally good as one taught exclusively by RL, while being faster to train and exhibiting, or at least having explored, a wider range of behaviour. Pseudo-code for the algorithm;

Optimize a population with EA as in algorithm 1.
Select candidate solutions from different niches.
for *n in candidate_solutions* **do**
  initialize policy as n **while** $i < burn\_in$ **do**
    run PPO optimization only updating valuenet parameters Θ but not policynet parameters θ
  end
  Optimize with PPO as in algorithm 2.
end

Algorithm 3: The Hybrid optimization procedure.

4.4 Programming Implementation Details

In order to maintain full low-level control and for portability reasons, a custom c++ neural network framework was built. Under the hood the Eigen linear algebra library [52] and AVX instruction set [53] were used to make computations efficient. A variety of evolutionary algorithm and reinforcement learning algorithms were then also implemented in c++, interfacing to these neural networks.
For the physic simulations the MuJoCo physics engine [54] was used. It offers "a unique combination of speed, accuracy and modeling power" [54] with a focus on contact dynamics. A body model can be defined in a hierarchical structure in an xml-file. This includes properties such as limbs, joints, tendons, ranges of motion, actuators, sensors, etc. A simulated body can then be controlled by inputting control values between -1 and 1 to each of the defined actuators. The strength of these actuators are limited by what is defined in the model. 32-bit floating point precision was used for all physics and network computations.
How to select a proper network architecture was another point of consideration. One approach was to allow the network structure to develop through evolutionary algo-

rithms [55]. However, lately the trend has only been to go 'deeper' with more and more internal nodes and layers, only limited by the ever increasing computational power at one's disposure. There was also the internal node structure to experiment with. New advancements such as ReLU activation functions [56] are gaining popularity while some swear by the old *tanh* activators. Parameter updating schemes were another point of contention, as the outdated standard stochastic gradient decent is being replaced by schemes such as RMSProp, Adagrad, and Adam [57][27]. Along with all this comes the generally large number of hyper-parameters requiring tuning in the neural network setting, with additional ones tacking on from the EA and RL optimization methods, resulting in a difficult to manage system that might be more art than science.
Other publications' hyper-parameters and network architectures were often used as starting points for our own experimentation. Sometimes tweaking a hyper-parameter has negligible effects while sometimes it is the difference between the algorithm working or doing nothing. In both these scenarios presenting results with respect to the tuning process is not of great interest, and we thus do not go into further details on them in our results. For the curiously inclined reader these hyper-parameters can be found in Appendix B.

5
Results

Here some case studies are presented on different simulation environments. Parameters used in these simulations can be found in Appendix B. Some case specific observations are made, with a more general discussion of the results following in Chapter 6.

5.1 Case study - Inverted double pendulum

The inverted double pendulum is a classical control problem, see Figure 5.1. It is not a locomotion problem per se, but it serves as a good benchmark for performance of different algorithms and parameters. The maximum force that can be generated by the cart is $F_{max} = 500\ N$ in either direction. Each pole has the same mass 4.1 kg and length 0.6 m. The cart has a mass of 9.4 kg. The gravity in the simulation is set to 9.82 m/s^2. To increase difficulty and add realism, the initial condition for the pole angles are distributed uniformly in the range $[-0.01, 0.01]$ rad, and motor noise of magnitude $F_{max} \cdot \delta$, where δ is white noise, $\delta \sim N(0, 0.001)$ sampled every time-step (intervals of 0.005 seconds).

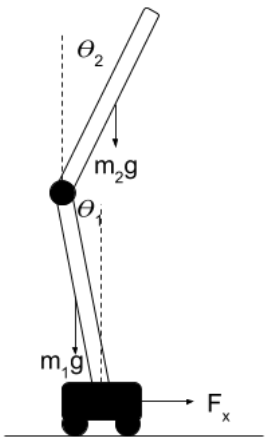

Figure 5.1: The inverted double pendulum is two poles connected with a hinge joint, balancing on a cart with a motor, which can provide acceleration horizontally. There are 2 rotation sensors for the pole joints measuring $\theta_1, \theta_2, \dot{\theta}_1, \dot{\theta}_2$, and sensors measuring position and velocity of the cart.

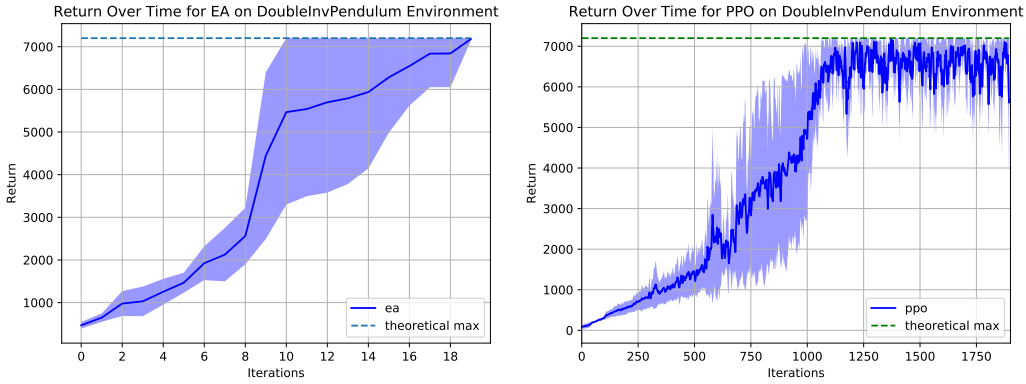

Figure 5.2: Return over the training procedure on the double inverted pendulum using EA (left panel) and RL (right panel). The shaded area shows one standard deviation when averaging over 5 runs of the learning algorithm. Both methods solve the problem well, reaching the theoretical maximum. Note that EA can solve the problem in a few iterations while PPO will always require a couple hundred iterations to converge even on easy problems.

The objective function R to maximize is chosen as

$$R = \sum r_t, \quad r_t = \left\{ \begin{array}{ll} h_{tip} - 0.03|x_{cart}|, & \text{for } h_{tip} \geq 0 \\ 0, & \text{for } h_{tip} < 0 \end{array} \right\} \tag{5.1}$$

evaluated over a maximum of $T = 30$ seconds or 6000 time-steps. h_{tip} is the height of the outer pole, and the evaluation is aborted if h_{tip} falls below 0, or if the cart moves to far sideways.
The result for training a controller to stabilize the inverted double pendulum can be seen in figure 5.2. Both EA and RL solve the task admirably, achieving the theoretical maximum return post training by balancing the pendulum until the time limit is reached.
The resulting controller can be seen in the link `https://www.youtube.com/watch?v=07XhpFIvzvQ&list=PLGbhmQSm6exr76oaHPLTd0V9L_3Btgy1Y`

5.2 Case study - Walker2d

The two-dimensional walker resembles the lower half of a simple humanoid like robot, see figure 5.3. It has a torso segment, and two legs separated into two sections, as well as large feet. The joints are given a human like range of motion. The entire body is squashed into two dimensions and stabilized in the third. As a consequence of the body being squashed the legs are allowed to phase through each other without colliding.
The action space consists of 6 muscle actuators controlling joint torques on the ankle-, knee-, and pelvic-joints. The state space consists of 21 dimensions; the 6 joints' positions and angular velocities (12 dimensions), and the torso segment of the body's z-position, x- and z-velocities, and rotation and rotational velocity around the y-axis (5 dimensions), and finally binary touch sensors on the heel and toe sections of the feet (4 dimensions).

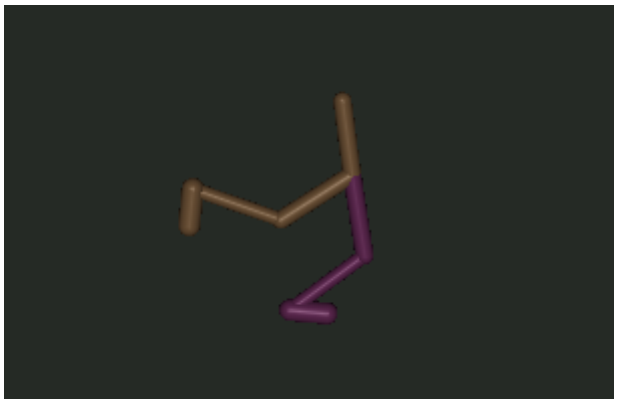

Figure 5.3: Screenshot of the 2d-walker environment simulated in MuJoCo.

The objective function R to maximize is chosen as

$$R = \sum r_t, \quad r_t = v - 10^{-5}c + 0.001. \tag{5.2}$$

Here v is forward velocity and the added control cost $c = \sum_i a_i^2$, where a_i is the control signal to actuator i, punishes excessively large control inputs. The final constant term 0.001 rewards agents for simply staying alive and not terminating early. This encourages balancing and staying upright during the early stages of learning where it can otherwise be difficult for agents to start doing anything worthwhile. Both this constant reward and the control cost are quickly overshadowed by the main velocity reward once the agent starts to get going. For example in this 2d-walker environment the trajectory return of a mediocre agent can be around 4000, with the control cost only contributing around -0.2 and the constant factor contributing around +2.
How the different algorithms performed on the task over their training period can be seen in Figure 5.4 and 5.5. For this task there are a multitude of very stable and well performing local minima involving both asymmetrical and symmetrical hopping behaviour that were especially prevalent in the EA optimization of Section 4.1. The asymmetrical behaviour typically consists of using one leg for forward propulsion through hopping and one leg flagging out backwards or forwards for stability and minor propulsion. The gaits are usually unique for different niches in the EA, depending on the merging radius of the niches. The resulting agents from EA optimization were outclassed in the long run by the PPO optimized agents of Section 4.2, which achieved larger returns when trained long enough. These agents usually reached around 7000 return with fairly stable running gaits.
Even better were the results from the hybrid optimization from Section 4.3, although this depends on which EA niche one uses as seed. As seen in Figure 5.5 there is one niche which ends up being far worse than the normal PPO optimization as the engraved behaviour is too strong. Our findings show that for the hybrid method, niches with symmetrical gaits with non-stiff legs produce the best results post PPO-optimization, even though they might not have been the top solutions found globally in the EA. The greatest returns were all achieved through the use of swinging both legs in a normal running gait, in accordance with what we as humans would consider natural looking movement.

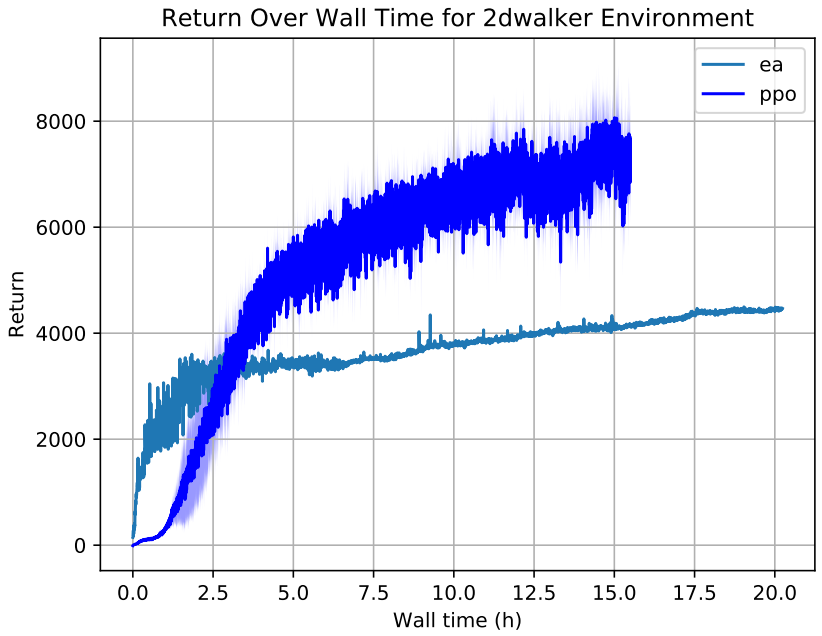

Figure 5.4: Comparison of EA and PPO optimization on the 2d walker environment with the same architecture (FFNN 64x32x32 as hidden layers) and initialization. This data is not smoothed, in contrast to that of figure 5.5.

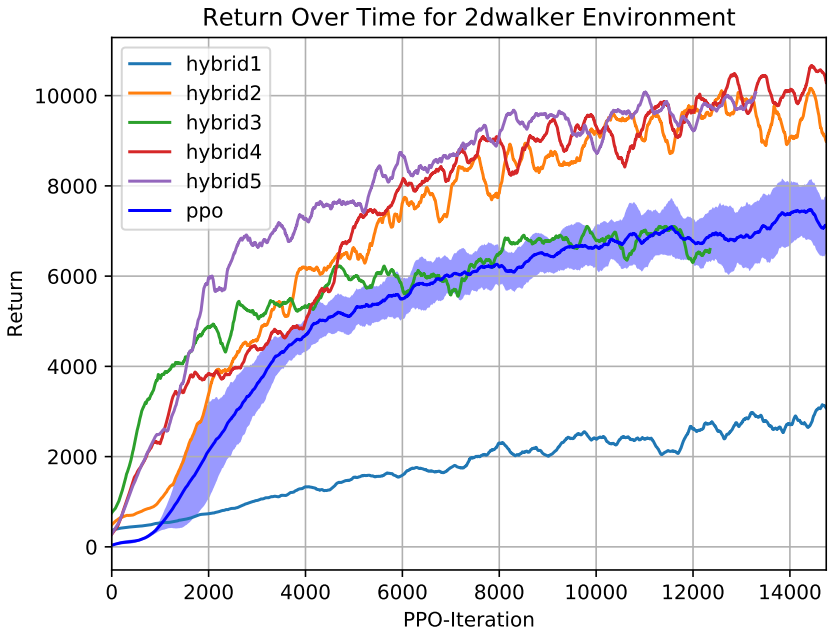

Figure 5.5: Average total return over a trajectory as a function of PPO training iterations. In dark blue is the normal PPO training mean, with its standard deviation over three runs shaded. The five hybrid runs are initialized from different EA-niches. Data points have been smoothed in 250 iteration intervals to make it easier to read. Note that PPO will always appear to be improving even towards the end, although this is just a result of the forced exploration being toned down and not actually improvements in behavior.

To best understand the different resulting gaits view them in video at `https://www.youtube.com/watch?v=Yn3KSd0CaNo&list=PLGbhmQSm6exrXwq4J2TffD6cQwTJjnbyc`

5.3 Case study - Quadruped

The quadruped is an insect-like robot consisting of four legs attached to a center sphere. The legs consist of two sections and are symmetrically placed around the body to provide a stable body by default, see Figure 5.6.
Each leg has two actuators, one controlling each joint, resulting in an eight-dimensional action space. The state space is 36-dimensional; positions and velocities of the 8 joints (16 dimensions), orientation and velocity of the body center (6 dimensions), touch sensors on the feet and center body (5 dimensions), accelerometer and gyroscope in the center body (6 dimensions), and three range finders facing forwards on the robot (3 dimensions). These range finders act as a crude form of vision, giving the distance to objects in a straight forward line.

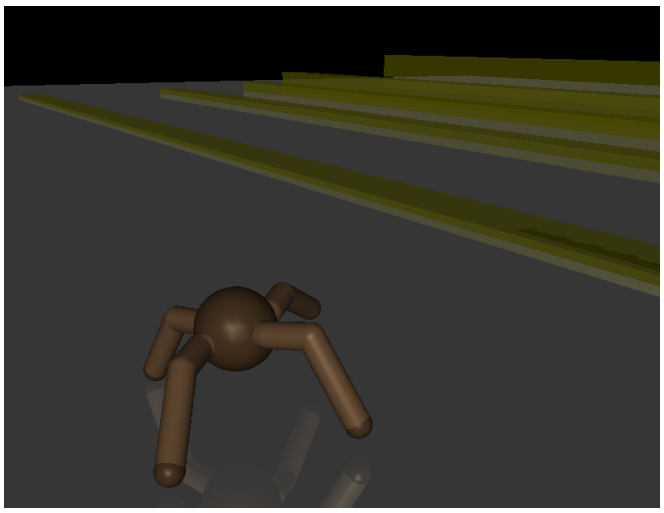

Figure 5.6: Screenshot of the quadruped environment simulated in MuJoCo. In (translucent) green some of the added obstacles are visible.

The robot is very stable by design so for added difficulty this case study included large rectangular boxes as obstacles. These boxes were placed in a configuration to increase the difficulty the further the robot got, with larger boxes and less space between them. The environment terminates early if the robot gets flipped over and lands on its back as it cannot flip itself back around.
The objective function R to maximize is chosen as

$$R = \sum r_t, \quad r_t = v_x - 10^{-5}c + 0.005 - 0.1|v_y|. \tag{5.3}$$

This reward function r_t is very similar to the 2d-walker's reward discussed in Section 5.2, but with slightly more constant reward incentivizing staying alive and not flipping over. There is also a slight punishment for moving sideways in the added third dimension to encourage movement in a straight line.

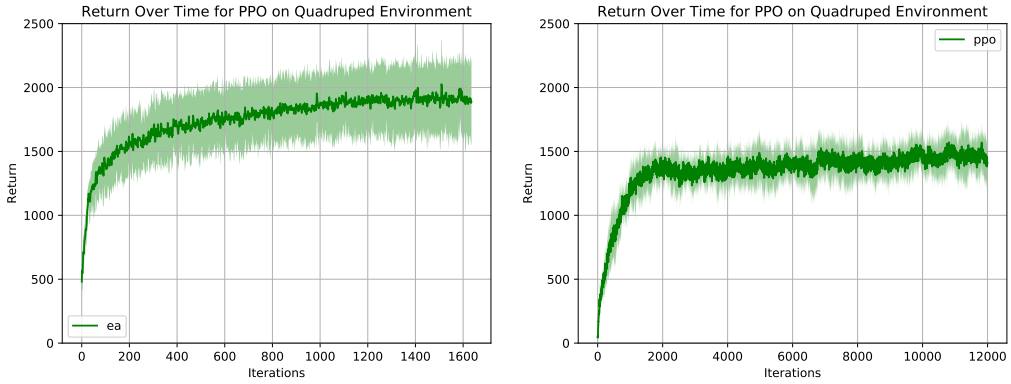

Figure 5.7: Return over the training procedure on the quadruped environment using EA on the left and RL on the right. The shaded area shows one standard deviation when averaging over multiple runs of the learning algorithm. EA achieves higher return on average, although its returns are slightly inflated due to agents getting lucky in some runs. RL averages an agents returns over a large amount of runs making it less prevalent.

The resulting agents displaced a variety of hopping behaviour. Traversing over an obstacle often leads to the robot rotating a bit and needing to reorient itself afterwards. All three methods achieved similar results on this task. The resulting gaits can be seen in video at `https://www.youtube.com/watch?v=HyVOP1FOMvI&list=PLGbhmQSm6exqB1dozO1boktEH1Leh4oP4`. Experiment parameters can be found in appendix B.

5.4 Case study - Humanoid

The humanoid robot has a simplified anatomy of a human, see Figure 5.8.
The action space is 21 dimensional, one actuator in each of the body's joints. The state space consists of 66 dimensions: positions and velocities of the joints (42 dimensions), four touch sensors on each foot (8 dimensions), touch sensors on the hands (2 dimensions), accelerometer and gyroscope in the head (6 dimensions), tendon velocities in the hip/knee connection tendons (2 dimensions), orientation centered on the pelvis (3 dimensions), and position of the head (3 dimensions).
The objective function R to maximize is chosen as

$$R = \sum r_t, \quad r_t = \left\{ \begin{array}{ll} v_x - 0.1|v_y| + 0.01 - 2 \cdot 10^{-5} c, & \text{for } h_{head} \geq 0.97 \\ 0, & \text{for } h_{head} < 0.97 \end{array} \right\} \tag{5.4}$$

evaluated over a maximum of 10 seconds or 2500 time-steps, and the evaluation is aborted if the head falls below half the human height.
This problem is arguably harder than any of the other cases. The humanoid struggles not to fall while still moving forward. The local optima are rich in nature. We observe limb flailing, almost running, jumping on one leg, kicking with one leg, jumping, pirouettes, side-skipping, ankle propelled jumping and many more. The results of the EA optimization from Section 4.1 and the RL optimization from Section 4.2 were even in terms of performance. Once again the hybrid method, Section

Figure 5.8: Screenshot of the humanoid simulated in MuJoCo.

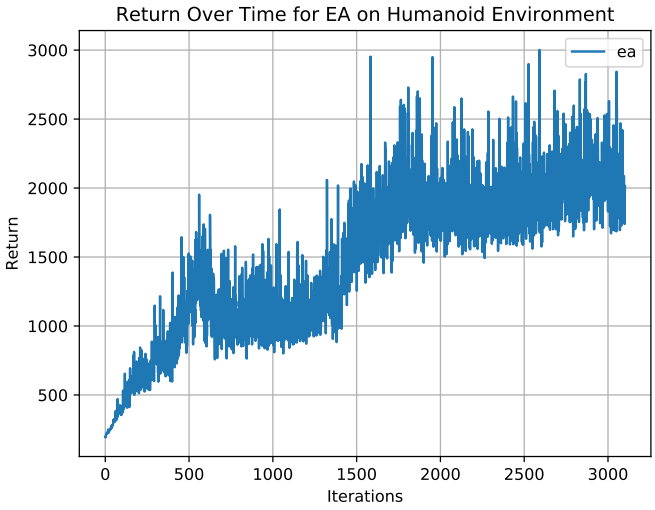

Figure 5.9: Return over the training period for EA run on the humanoid environment. Note that some of the 'spikes' just correspond to an agent getting lucky on a run. These best agents remain in the population unaltered, due to elitism, but on subsequent runs they usually will not repeat their previous lucky performance.

4.3, achieved the best results, although there is still a lot of room for improvement. A curation of the resulting gaits can be seen in video at `https://www.youtube.com/watch?v=vC50AJ_nR9E&list=PLGbhmQSm6exp1eUSIQpH81OOsvAcwBlvn`. Training curves are show in Figures 5.9 and 5.10. Experiment parameters can be found in appendix B.

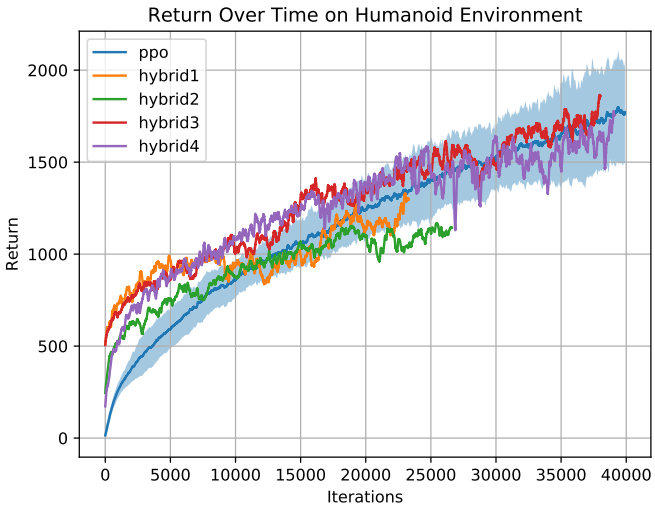

Figure 5.10: Return over the training period for PPO and the hybrid method run on the humanoid environment. The shaded region shows one standard deviation over five runs of PPO. Smoothed in 100 step intervals for ease of readability. On this problem the hybrid method does not appear to grant any extra performance over normal PPO.

6
Discussion

6.1 Summary of the results

The results of the two simpler environments (the inverted double pendulum and 2-dimensional biped) were satisfactory, while the more difficult environments leave room for improvement. The humanoid, in particular, proved to be very difficult to control in a stable manner.
Some general observations which hold true over all the case studies follow. The EA method excels in producing a large variety of gaits, preserving the diversity of its solutions over the niches. To the human eye it was often evident that these solutions were not fully developed and optimized. This is in contrast to RL where the solutions often appeared more optimized but lacked in variety. The hybrid method, Section 4.3, ended up out-performing both the other methods across all environments, and can be considered the true result of this work.
Our aim of being able to derive locomotive controllers for general bodies was partially realized. The hybrid method is capable of producing controllers for unstructured locomotion problems, but does not scale to the more difficult problems as well as we had hoped. Our main issue was the difficulty of changing behavioural regime, to be discussed in Section 6.2. Currently the resulting gaits can not compete with those from traditional autonomous control approaches such as in Reference [1]. Whether this can be remedied by further increases in computational power, algorithm improvements, and overall maturing of the machine learning field is difficult to say, but judging by the last few years of progress we believe this to be the case.
Comparing to other work within machine learning and locomotion our results are equal or slightly inferior as a result of handling a more generalized problem, and using less reward engineering than for example Reference [10] (to be discussed in Section 6.3).

6.2 The difficulty of changing behavioural regime

An underlying issue for all the optimization methods discussed in this work is the difficulty of escaping local minima. The optimization is performed in a way that seeks[1] continuous improvement of the agent's behaviour, leading to a 'path' of better performing behaviour and culminating in some final gait which seemingly cannot be

[1] In reality the learning process is very noisy as seen in the return over time graphs, but the aim is constant improvement

improved upon much further. The final gait is thus highly dependant on the chosen path leading up to it. For example a biped robot's path of improvement might look something like in figure 6.1. As illustrated, the agent might get entrenched in a more attractive local optima early on, leading to suboptimal final performance.

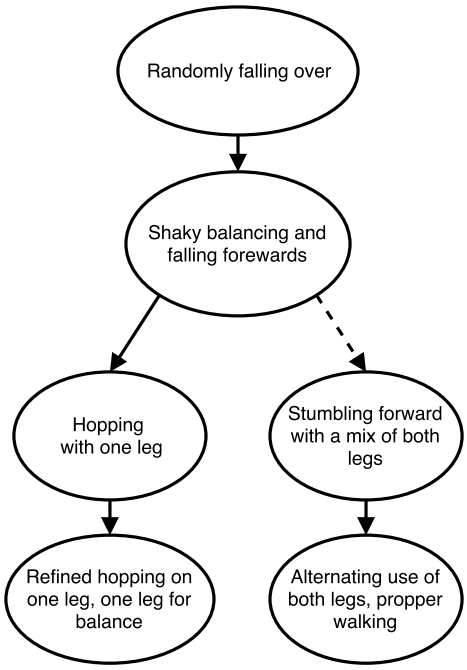

Figure 6.1: A hypothetical improvement path for a biped robot. It is difficult to reach the global optima (bottom right) as the alternative path is more attractive during early optimization.

With RL the more complex the environment the more evident this issue becomes. For example in the case of the 2d-walker most PPO trained agents ended up with similar final behaviour. The behaviour early on did not appear to define the final behaviour. Meanwhile with the humanoid environment it is easy to see similarities between the agent's behaviour early on and its final behaviour. In order to mitigate this problem it would be nice to force more long term exploration, correlated over multiple timesteps, but it is not clear how to go about this in a scalable way.
With EAs on the other hand this phenomenon is somewhat desired, with different niches exhibiting different types of behaviour. At the same time it would be preferable for larger amounts of new niches to appear over time, corresponding to larger behavioural shifts. Currently the niches are fairly static with entrenched behaviour seldom giving way for larger behavioural regime changes.

6.3 Reward Function Engineering

The reward/objective function plays an important role in judging the agents and ultimately determining the result. For this reason it can be a powerful tool to guide the learning process by hand designing the reward function to encourage or discourage specific states, forcing the desired behaviour. For example in a biped

one could punish lifting feet too high, swinging arms too much, etc. The problem is that reward engineering introduces a very human-dependent component to the process. With it comes a lot of preconceived notions about the solution, which constrains the natural 'artificial intelligence' approach. Hypothetically one could force any solution to be the 'optimal' one with sufficient reward engineering. The environments' terminating conditions also play a similar role to a smaller degree, and can also be used in the same way to engineer the results.
For all our case studies we have made use of very simple reward functions. The reward is proportional to forward velocity, with slight punishment to excessively large control inputs and sideways velocity, as well as a small constant reward to discourage early termination. Along with this, the termination conditions have been set to only detect states in which the agent is beyond saving, such as the humanoid falling over or quadruped being upside down. This is a general form of constructing a locomotion problem environment in which you could place any kind of robot body without any preconceived notions about the resulting gait. In this way agents are less guided in solving the problem. In practical applications the amount of reward engineering and artificial guidance desired will depend on the problem, but the goal of this work was to be as general as possible and avoid focusing on such aspects.

6.4 The Synergistic Nature of RL and EA

There are multiple reasons that RL and EA works well together as a hybrid method for the tasks presented in this work. First and foremost EA operates on a coarser temporal scale; an individual is only judged and assigned a fitness depending on the total trajectory return. In contrast to this, RL breaks the trajectory down into small pieces and tries to look at each individual action performed. Every action is judged and parameters are tweaked in a more refined way, albeit from an estimated and noisy error signal.
Secondly, on a simplified fundamental level, RL wanders through parameter-space using stochastic gradient descent, optimizing an individual agent. In contrast EA is more about optimizing an entire population, or a distribution of agents, also discussed in Reference [58]. The result is two methods that behave very differently depending on the "fitness" landscapes, the solution performance as a function of the parameters that are optimized. The EA population will be distributed among the peaks in the landscape, and if the peaks are very narrow any large mutation or crossover (or recombination) will most likely be "bad". In other words, individuals subject to crossover will have to populate the same narrow region which means less novelty. The occurrence of any significant leaps in the fitness landscape in that region will be rare, and progress will be very slow. In contrast to this RL will be able to determine which direction to move in the same region and keep progressing. On the other hand if there is a large enough "ridge" so that we have to make a jump in order for the neural network to not perform very poorly, RL will struggle since it moves almost continuously in small steps across the parameters space.
Illustrative examples of how the algorithms tend to traverse the parameter space can be seen in figures 2,4,5, and 6 in [58]. This also leads to EA achieving a broader spectrum of behaviour, whereas RL often appears to follow more consistent paths during

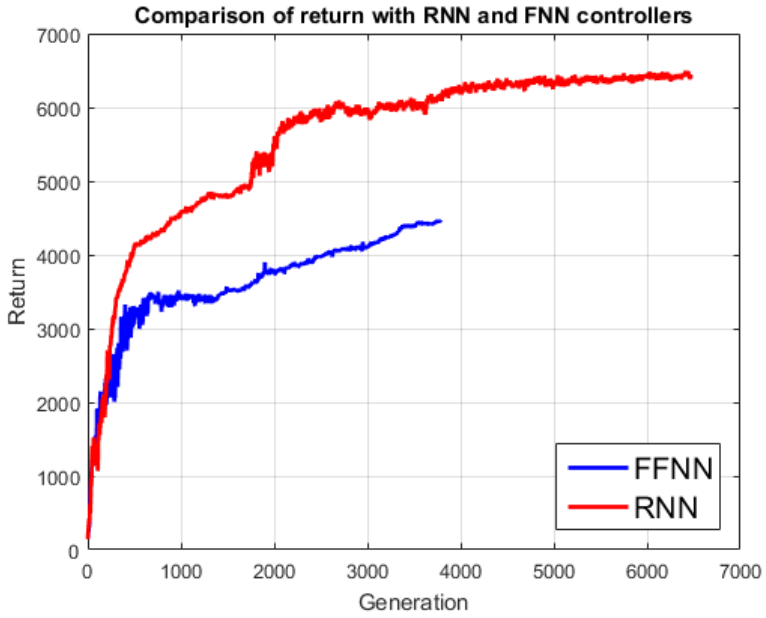

Figure 6.2: A recurrent (RNN) and a feed-forward network (FFNN) being trained with EA on the 2d-walker environment of 5.2. By adding a recurrent layer with 16-tanh nodes, the same algorithm was able to produce better controllers.

optimization. It is important to note that this 'broader spectrum of behaviour' will of course consist of many inferior local optima, but nevertheless this broader exploration might just be of use for RL. The classical trade off of exploration-exploitation always makes itself known.
On a practical level, EA is far better suited for parallelization as each individual agent can be simulated and updated independently, with some synchronization between generations. On the other hand PPO is not as straight forward to parallelize because although simulations of trajectories can be done independently the parameter updates cannot. On top of this there is EA's ability to quickly find solutions in simple problems such as the inverted pendulum, while RL always takes a while to converge. EAs thus seems to have an advantage when it comes to scalability.
All these reasons encourage the idea of performing a 'pre-optimization' through EA followed by RL for refinement. Alternatively one could view it as RL, with policy initialization performed by EA. As seen in Section 5.2 this hybrid method has potential of reaching better optima, achieving larger returns than either RL or EA did on their own. At the same time it can also perform worse, as evident by *hybrid1* in figure 5.3.

6.5 Significance of Network Architectures

Activation functions and updating schemes are hard to evaluate, but usually will not make or break the whole algorithm. They play a larger role in traditional classification tasks where small increments of performance are of huge importance. In RL and EA the learning process is very convoluted, and there is already so much

noise in the learning signals that these choices are not of equal importance.
Architectural modifications, such as adding recurrent layers can have larger effects, see Figure 6.2 for some initial experimentation. RNNs have been found to be well suited for continuous time domains such as control and speech recognition [23, 59]. The problem with them is training, either due to chaotic behaviour in regions of the parameters space, vanishing/exploding gradients and increased computational complexity in gradient descent methods. Further work could examine the effect of using RNNs for this task, as they show promising performance in a few tests.

6.6 Concluding Thoughts

The cross section of AI and locomotion is an interesting and complex problem. In this work three main methods have been applied to allow simulated agents to learn how to traverse their environments. We have found that these different methods have their own strengths and weaknesses.
Evolutionary algorithms find a wide variety of decent solutions regardless of the problem, but are perhaps too rudimentary to refine these solutions to their maximum potential. Their main strengths are the diversity they achieve and their ease of scalability in a computational sense.
Reinforcement learning, or the Proximal Policy Optimization algorithm to be more exact, achieves exceptional results when it does work. However getting to that point requires a bit of toying around with hyperparameters or the algorithm often finds itself stuck early on in the optimization procedure. With higher dimensional problems we also had difficulty getting the algorithm to reach stable gaits, although perhaps this can be mitigated by just adding more computational power.
Finally we tried a hybrid method, utilizing EA optimization first to find population niches from which to initialize PPO optimization. This approach seemed to give a nice boost to the PPO results in the problems where PPO did well, but did not help much when PPO was already struggling, as in the humanoid environment.
As discussed in 6.2 a common issue for all the methods are how the optimization tends to follow a certain behavioural path during the learning process, and will sometimes get stuck in local optima due to inferior paths being more attractive. Further work could focus on how to solve this issue.

Bibliography

[1] Jacob Reher, Eric A Cousineau, Ayonga Hereid, Christian M Hubicki, and Aaron D Ames. Realizing dynamic and efficient bipedal locomotion on the humanoid robot durus. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 1794–1801. IEEE, 2016.

[2] Daniel Holden, Taku Komura, and Jun Saito. Phase-functioned neural networks for character control. *ACM Transactions on Graphics (TOG)*, 36(4):42, 2017.

[3] Thomas Geijtenbeek, Michiel van de Panne, and A. Frank van der Stappen. Flexible muscle-based locomotion for bipedal creatures. *ACM Transactions on Graphics*, 32(6), 2013.

[4] D Casey Kerrigan, Mary K Todd, Ugo Della Croce, Lewis A Lipsitz, and James J Collins. Biomechanical gait alterations independent of speed in the healthy elderly: evidence for specific limiting impairments. *Archives of physical medicine and rehabilitation*, 79(3):317–322, 1998.

[5] Randall D Beer, Hillel J Chiel, Roger D Quinn, Kenneth S Espenschied, and Patrik Larsson. A distributed neural network architecture for hexapod robot locomotion. *Neural Computation*, 4(3):356–365, 1992.

[6] Vinod K Valsalam and Risto Miikkulainen. Modular neuroevolution for multilegged locomotion. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 265–272. ACM, 2008.

[7] Brian F Allen and Petros Faloutsos. Evolved controllers for simulated locomotion. In *MIG*, pages 219–230. Springer, 2009.

[8] José Santos and Ángel Campo. Biped locomotion control with evolved adaptive center-crossing continuous time recurrent neural networks. *Neurocomputing*, 86:86–96, 2012.

[9] Xin Deng, Jian-Xin Xu, Jin Wang, Guo-yin Wang, and Qiao-song Chen. Biological modeling the undulatory locomotion of c. elegans using dynamic neural network approach. *Neurocomputing*, 186:207–217, 2016.

[10] Nicolas Heess, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, Ali Eslami, Martin Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[11] Ari Weinstein and Matthew M Botvinick. Structure learning in motor control: A deep reinforcement learning model. *arXiv preprint arXiv:1706.06827*, 2017.

[12] Mattias Wahde. *Biologically inspired optimization methods: an introduction*. WIT press, 2008.

[13] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[14] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[16] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1889–1897, 2015.

[17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[18] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *International Conference on Machine Learning*, pages 1329–1338, 2016.

[19] Wulfram Gerstner. Hebbian learning and plasticity. *From neuron to cognition via computational neuroscience*, pages 0–25, 2011.

[20] Larry F Abbott and Sacha B Nelson. Synaptic plasticity: taming the beast. *Nature neuroscience*, 3(11s):1178, 2000.

[21] Ramakrishnan Iyer, Vilas Menon, Michael Buice, Christof Koch, and Stefan Mihalas. The influence of synaptic weight distribution on neuronal population dynamics. *PLoS computational biology*, 9(10):e1003248, 2013.

[22] Simon S Haykin, Simon S Haykin, Simon S Haykin, and Simon S Haykin. *Neural networks and learning machines*, volume 3. Pearson Upper Saddle River, NJ, USA:, 2009.

[23] Danilo P Mandic, Jonathon A Chambers, et al. *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. Wiley Online Library, 2001.

[24] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806, 1993.

[25] Joel Lehman, Jay Chen, Jeff Clune, and Kenneth O Stanley. Safe mutations for deep and recurrent neural networks through output gradients. *arXiv preprint arXiv:1712.06563*, 2017.

[26] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[27] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[28] Alden H Wright. Genetic algorithms for real parameter optimization. In *Foundations of genetic algorithms*, volume 1, pages 205–218. Elsevier, 1991.

[29] Evert Haasdijk, Andrei A Rusu, and AE Eiben. Hyperneat for locomotion control in modular robots. In *International Conference on Evolvable Systems*, pages 169–180. Springer, 2010.

[30] Luke Temby, Peter Vamplew, and Adam Berry. Accelerating real-valued genetic algorithms using mutation-with-momentum. In *Australasian Joint Conference on Artificial Intelligence*, pages 1108–1111. Springer, 2005.

[31] Kay Chen Tan, Tong Heng Lee, and Eik Fun Khor. Evolutionary algorithms with dynamic population size and local exploration for multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 5(6):565–588, 2001.

[32] Shouyong Jiang and Shengxiang Yang. A steady-state and generational evolutionary algorithm for dynamic multiobjective optimization. *IEEE Transactions on Evolutionary Computation*, 21(1):65–82, 2017.

[33] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[34] Scott Sanner and Craig Boutilier. Approximate linear programming for first-order mdps. *arXiv preprint arXiv:1207.1415*, 2012.

[35] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering, 1994.

[36] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[37] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[38] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[39] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[40] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[41] Michael T Rosenstein, Andrew G Barto, Jennie Si, Andy Barto, Warren Powell, and Donald Wunsch. Supervised actor-critic reinforcement learning. *Handbook of learning and approximate dynamic programming*, pages 359–380, 2004.

[42] Frank Veenstra, Alexander Struck, and Matthias Krauledat. Acquiring efficient locomotion in a simulated quadruped through evolving random and predefined neural networks. In *Artificial Evolution*, 2015.

[43] Danilo Vasconcellos Vargas and Junichi Murata. Spectrum-diverse neuroevolution with unified neural models. *IEEE transactions on neural networks and learning systems*, 28(8):1759–1773, 2017.

[44] Pedro Silva, Cristina P Santos, Vítor Matos, and Lino Costa. Automatic generation of biped locomotion controllers using genetic programming. *Robotics and Autonomous Systems*, 62(10):1531–1548, 2014.

[45] Tim Salimans, Jonathan Ho, Xi Chen, and Ilya Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[46] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Deep neuroevolution: Genetic algorithms are

a competitive alternative for training deep neural networks for reinforcement learning. *arXiv preprint arXiv:1712.06567*, 2017.

[47] Joel Lehman and Kenneth O Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 211–218. ACM, 2011.

[48] Andrés Espinal, Horacio Rostro-Gonzalez, M Carpio, Erick Israel Guerra-Hernandez, M Ornelas-Rodriguez, HJ Puga-Soberanes, Marco Aurelio Sotelo-Figueroa, and Patricia Melin. Quadrupedal robot locomotion: a biologically inspired approach and its hardware implementation. *Computational intelligence and neuroscience*, 2016, 2016.

[49] Nick Jakobi, Phil Husbands, and Inman Harvey. Noise and the reality gap: The use of simulation in evolutionary robotics. In *European Conference on Artificial Life*, pages 704–720. Springer, 1995.

[50] Deepti Gupta and Shabina Ghafir. An overview of methods maintaining diversity in genetic algorithms.

[51] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

[52] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[53] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 19:20, 2008.

[54] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[55] Xin Yao and Yong Liu. A new evolutionary system for evolving artificial neural networks. *IEEE transactions on neural networks*, 8(3):694–713, 1997.

[56] George E Dahl, Tara N Sainath, and Geoffrey E Hinton. Improving deep neural networks for lvcsr using rectified linear units and dropout. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8609–8613. IEEE, 2013.

[57] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[58] Joel Lehman, Jay Chen, Jeff Clune, and Kenneth O Stanley. Es is more than just a traditional finite-difference approximator. *arXiv preprint arXiv:1712.06568*, 2017.

[59] Barak A Pearlmutter. Learning state space trajectories in recurrent neural networks. *Neural Computation*, 1(2):263–269, 1989.

A

Parameter Derivatives of the Multivariate Gaussian

With policy π as an n-dimensional multivariate Gaussian with diagonal Σ matrix:

$$\pi_\theta(\mathbf{a_t}|\mathbf{s_t}) = \frac{1}{(2\pi)^{n/2}\sigma_1\sigma_2 \cdot ... \cdot \sigma_n} \exp\left(-\frac{(a_{t,1}-\mu_1)^2}{2\sigma_1^2} - \frac{(a_{t,2}-\mu_2)^2}{2\sigma_2^2} - ... - \frac{(a_{t,n}-\mu_n)^2}{2\sigma_n^2}\right)$$

$$\begin{aligned}
\frac{\partial}{\partial\sigma_i}\pi_\theta(\mathbf{a_t}|\mathbf{s_t}) &= \frac{1}{(2\pi)^{n/2}}\left(\frac{1}{\sigma_1 \cdot ... \cdot (-\sigma_i^2) \cdot ... \cdot \sigma_n}\exp(...) + \frac{1}{\sigma_1 \cdot ... \cdot \sigma_n}\exp(...)\left(\frac{2(a_{t,i}-\mu_i)^2}{2\sigma_i^3}\right)\right) = \\
&= \frac{1}{(2\pi)^{n/2}}\frac{1}{\sigma_1\sigma_2 \cdot ... \cdot \sigma_n}\exp(...)\left(\frac{1}{-\sigma_i} + \frac{(a_{t,i}-\mu_i)^2}{\sigma_i^3}\right) = \\
&= \pi_\theta(\mathbf{a_t}|\mathbf{s_t}) \cdot \left(\frac{(a_{t,i}-\mu_i)^2}{\sigma_i^3} - \frac{1}{\sigma_i}\right) = \\
&= \pi_\theta(\mathbf{a_t}|\mathbf{s_t}) \cdot \frac{1}{\sigma_i}\left(\left(\frac{a_{t,i}-\mu_i}{\sigma_i}\right)^2 - 1\right)
\end{aligned}$$

The mean is straight forward

$$\frac{\partial}{\partial\mu_i}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \cdot \left(\frac{a_{t,i}-\mu_i}{\sigma_i^2}\right)$$

B

Hyper-Parameters Used in Case Study Simulations

Neuralnet layers	4
NN size Policy	6, 32, 16, 1
Activation functions	tanh,tanh,tanh
initialization	Xavier
timestep_delta_t	0.005
max_timesteps simulated	6000
framskips	0
—**RL specific**—	
Parameter updating scheme	Adam
Learning rate α	0.0001
Adam_epsilon	10^{-6}
Adam_b1	0.9
Adam_b2	0.999
NN size ValueFunction	6, 32, 16, 1
PPO_max_iterations	5000
PPO_traj_per_batch	16
PPO_minibatch_size	64
PPO_epochs_per_update	1
PPO_epsilon	0.2
PPO_Sigma_decay	Exp-decay from 0.5 to 0.2
reward_decay γ	0.995
—**EA specific**—	
initial population size	128 individuals
max niches	100
r_{merge}	3
r_{niche}	4
$p_{mutation}$	0.01
initial creep mutation σ_j	1.0
max gene mutation moment m_j	$m_j < \sigma_j/10$
$p_{crossover}$ inside niche	0.6
$p_{crossover}$ global	0.01
tournament size in niche	8
tournament size globally	200
p_{tour}	0.6
elitism count	4
number of parents in multiparent crossover	4
linear replacement start index	20
extinction size	10000

Table B.1: Parameters used for the double inverted pendulum environment.

Neuralnet layers	5
NN size Policy	21, 64, 32, 32, 6
Activation functions	tanh,tanh,tanh,tanh
initialization	Xavier
timestep_delta_t	0.005
max_timesteps simulated	2000
framskips	0
—**RL specific**—	
Parameter updating scheme	Adam
Learning rate α	$5 \cdot 10^{-5}$
Adam_epsilon	10^{-6}
Adam_b1	0.9
Adam_b2	0.999
NN size ValueFunction	21, 64, 32, 32, 1
PPO_max_iterations	15000
PPO_traj_per_batch	32
PPO_minibatch_size	64
PPO_epochs_per_update	1
PPO_epsilon	0.2
PPO_Sigma_decay	Exp-decay from 0.5 to 0.2
reward_decay γ	0.995
—**EA specific**—	
initial population size	128 individuals
max niches	100
r_{merge}	3
r_{niche}	4
$p_{mutation}$	0.01
initial creep mutation σ_j	1.0
max gene mutation moment m_j	$m_j < \sigma_j/10$
$p_{crossover}$ inside niche	0.6
$p_{crossover}$ global	0.01
tournament size in niche	8
tournament size globally	200
p_{tour}	0.6
elitism count	4
number of parents in multiparent crossover	4
linear replacement start index	20
extinction size	10000
—**Hybrid specific**—	
Burn in time	1000

Table B.2: Parameters used for the walker2d environment.

Neuralnet layers	5
NN size Policy	36, 64, 48, 32, 8
Activation functions	tanh,tanh,tanh,tanh
initialization	Xavier
timestep_delta_t	0.01
max_timesteps simulated	2500
framskips	0
—**RL specific**—	
Parameter updating scheme	Adam
Learning rate α	$5 \cdot 10^{-5}$
Adam_epsilon	10^{-6}
Adam_b1	0.9
Adam_b2	0.999
NN size ValueFunction	36, 64, 48, 32, 1
PPO_max_iterations	12000
PPO_traj_per_batch	32
PPO_minibatch_size	64
PPO_epochs_per_update	1
PPO_epsilon	0.2
PPO_Sigma_decay	Exp-decay from 0.5 to 0.2
reward_decay γ	0.995
—**EA specific**—	
initial population size	128 individuals
max niches	100
r_{merge}	3
r_{niche}	4
$p_{mutation}$	0.01
initial creep mutation σ_j	1.0
max gene mutation moment m_j	$m_j < \sigma_j/10$
$p_{crossover}$ inside niche	0.6
$p_{crossover}$ global	0.01
tournament size in niche	8
tournament size globally	200
p_{tour}	0.6
elitism count	4
number of parents in multiparent crossover	4
linear replacement start index	20
extinction size	10000
—**Hybrid specific**—	
Burn in time	1000

Table B.3: Parameters used for the quadruped environment.

Neuralnet layers	5
NN size Policy	66, 96, 64, 48, 21
Activation functions	tanh,tanh,tanh,tanh
initialization	Xavier
timestep_delta_t	0.004
max_timesteps simulated	2000
framskips	0
—**RL specific**—	
Parameter updating scheme	Adam
Learning rate α	$5 \cdot 10^{-5}$
Adam_epsilon	10^{-6}
Adam_b1	0.9
Adam_b2	0.999
NN size ValueFunction	66, 96, 64, 48, 1
PPO_max_iterations	40000
PPO_traj_per_batch	32
PPO_minibatch_size	64
PPO_epochs_per_update	1
PPO_epsilon	0.2
PPO_Sigma_decay	Exp-decay from 0.5 to 0.2
reward_decay γ	0.995
—**EA specific**—	
initial population size	128 individuals
max niches	100
r_{merge}	3
r_{niche}	4
$p_{mutation}$	0.01
initial creep mutation σ_j	1.0
max gene mutation moment m_j	$m_j < \sigma_j/10$
$p_{crossover}$ inside niche	0.6
$p_{crossover}$ global	0.01
tournament size in niche	8
tournament size globally	200
p_{tour}	0.6
elitism count	4
number of parents in multiparent crossover	4
linear replacement start index	20
extinction size	10000
—**Hybrid specific**—	
Burn in time	1000

Table B.4: Parameters used for the humanoid environment.