A Neuroeovolution Approach to Robotic Arm Control

Anthony Horgan
School of Computer Science
National University of Ireland Galway

Supervisor(s)
Dr. Karl Mason

In partial fulfillment of the requirements for the degree of
MSc in Computer Science (Artificial Intelligence)
September 7, 2022

DECLARATION I, Anthony Horgan, hereby declare that this thesis, titled "A Neuroeovolution Approach to Robotic Arm Control", and the work presented in it are entirely my own except where explicitly stated otherwise in the text, and that this work has not been previously submitted, in part or whole, to any university or institution for any degree, diploma, or other qualification.

Signature: _____

Abstract

This project investigates an application of neuroevolution for control of a robotic arm. The main focus of this project is on exploring best methods of evolving the weights of a neural network for movement control of a robotic arm with six degrees of freedom. Different control techniques are evaluated and compared. A secondary focus is on investigating the usefulness of expert demonstrations for evolving controllers. The most successful neuroevolution approach produces a network controller that can position the end effector of the arm to within 1.5 cm of a randomly placed target.

Keywords: Neuroevolution, Artiicial Intelligence, Reinforcement Learning, Neural Networks, Robot Control

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Robotic arms are used to perform an extensive range of tasks in a number of real-world environments including farms [1], warehouses [2, 3] and the manufacturing industry [4, p. 969-974]. In theory they can be manually programmed to perform an infinite number of tasks. Tasks which are too dangerous, require extreme precision and accuracy or are too laborious or tedious for humans can be performed quickly and safely by robotic arms. The use of robotic arms in factories can streamline and speed up production with improved accuracy over human workers. Robotic arms are used for welding, machining, cutting and assembling. In warehouses and factories, robotic arms are used to automatically pick up and place objects. Traditionally, these robots needed to be carefully programmed by experts to perform narrowly-defined tasks.

With modern advances in agent-based systems and agent learning, it is possible for autonomous agents to learn to perform complex tasks. Modern artificial intelligence (AI) agents learn useful behaviours in a wide variety of environments with only a small amount of human oversight. By applying learning agents to non-

trivial tasks, the agents are free to quickly learn and evaluate complex behaviours. The power of modern techniques can allow agents to extract and recognise complex patterns far beyond the capabilities of humans. Learning agents are not influenced by human design biases and practices and so can produce innovative, efficient and novel solutions to complex problems. AI techniques - specifically, machine learning - can be utilized to create intelligent agents which can adapt to dynamic environments (something that is far beyond the reach of manually programmed robots). A robot can be programmed to perform a specific task (e.g. pick up an object and place it in a new location), but a single solution is applicable only to a single, specific environment. With any small change in the environment (e.g. change in object location, change in target location, new obstacles etc.), the robot needs to be reprogrammed. AI can be used to develop a controller which can easily adapt to dynamic environments. With the rise of automation in the workplace [5, 6], improvement and development of intelligent, autonomous robotic controllers are more important than ever.

Neuroevolution is a technique which is often used to solve RL problems. Neuroevolution combines evolutionary algorithms (EA) with artificial neural networks (NN). Simple techniques use popular evolutionary algorithms to evolve the weight parameters of the network. More advanced neuroevolution algorithms can evolve both the network parameters and the actual architecture (topology) of the network itself.

While Deep reinforcement learning (DRL) is one of the most well-studied approaches to robotic arm control [7, 8, 9], researchers have been looking towards neuroevolution as an alternative which solves some of the issues with DRL, namely difficulty in dealing with sparse rewards and long time horizons as well as choosing an appropriate network architecture [10]. Neuroevolution has been shown to be a competitive alternative for training deep NNs for reinforcement learning tasks [11,

12]. Methods such as Deep Q-Learning have been demonstrated to be effective techniques for developing robotic arm controllers [9]. It is hypothesised that neuroevolution can also demonstrate impressive performance on similar robotic control tasks.

Neuroevolution has previously been used for movement control of a robotic arm [13, 14, 15, 16]. Many of these approaches use neuroevolution algorithms such as NEAT which evolve the network topology [13, 15, 16]. This work focuses on using EAs to evolve the weights of NNs with simple architectures to produce controllers capable of accurately controlling a robotic arm to perform a complex task. This work implements state of the art evolutionary methods in a modern and highly realistic robotic simulator. This work also investigates the impact of expert demonstrations on neuroevolution of a robotic arm controller. There are no examples of this in the existing literature.

1.2 Purpose

This research aims to investigate how best to apply neuroevolution to robotic arm control. The main goal of this research is to determine to what extent neuroevolution is a viable alternative to movement control of a robotic arm.

1.3 Research Questions

1. How should neuroevolution be implemented to effectively evolve a robotic arm controller?

2. How does expert demonstration impact the performance of the evolved controller?

Chapter 2

Background and Related Work

2.1 Reinforcement Learning

Reinforcement Learning (RL) is one of the main paradigms of machine learning. RL can train an autonomous agent (computer program capable of independent, intelligent action in an environment) to perform tasks by rewarding desired behaviours and punishing undesired ones. In RL, an agent learns to perform a task in an environment. The agent receives information on the state of the environment, uses this information to perform an action on the environment (thus changing its state) and receives a reward telling the agent how good or bad the action was. Through trial and error, the agent learns which actions to perform in which states in order to maximize reward. In order for an agent to learn and improve, it must explore new actions and also exploit actions which are known to be good. This exploration-exploitation trade-off is a ubiquitous dilemma in the field of RL. Several different RL techniques and algorithms have been used to solve robotic control tasks [17, 8, 18].

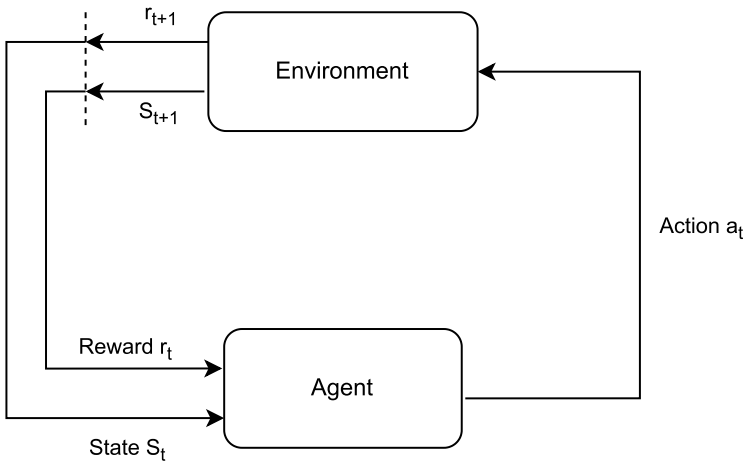

Figure 2.1: Visualization of how an agent interacts with an environment

2.2 Neural Networks

Deep Learning (a subset of machine learning) is a powerful approach which uses neural networks to solve complex, unstructured problems. Input data is processed by several layers of nodes. Each node has learnable weight parameters. In each node, the outputs from nodes in the preceding layer are combined in a weighted sum (according to the weight parameters), this sum is passed through a non-linear activation function to produce the output value for that node. NNs are usually trained using backpropagation with gradient descent. At every iteration, each weight is updated according to its gradient. The gradient of a weight is the derivative of the loss function (which is a measure of how well the NN is performing at the task) with respect to that weight. Deep neural networks with many layers can be trained in this way (for many iterations) until they can accurately approximate extremely complex functions. These deep networks can learn patterns and extract useful information from the input data to intelligently solve difficult problems.

With recent advances in computational resources and research on NN archi-

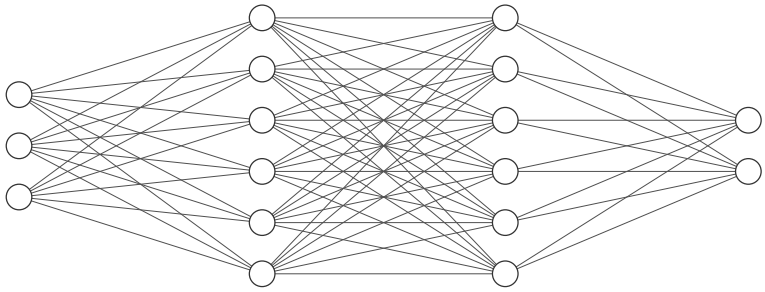

Figure 2.2: Neural Network with three input nodes, two hidden layers with six nodes each and two output nodes

tectures, NNs have become a powerful tool for solving classification, information extraction and data generation tasks as well as a plethora of other real-world problems. From playing board games such as Go at a superhuman level [19], to outperforming humans in all fifty games in the Atari 2600 benchmark [20], to developing continuous robotic controllers [21, 9], deep NNs have been shown to be powerful tools for tackling RL problems.

2.3 Evolutionary Algorithms

Evolutionary algorithms (EA) are a set of metaheuristic techniques inspired by natural evolution, which are used to solve optimization problems [22]. EAs are made up of a population of individuals (also referred to as candidate solutions), each individual represents one possible solution to the optimization problem. Each individual has a genotype (genetic information made up of a number of genes) and a phenotype (physical manifestation of genotype). The population is initialized randomly. Individuals are evaluated according to a fitness function which measures how good the individual is at solving the optimization problem. The fitness function is also refferred to as the objective function or cost function (with fitness, high scores are better and with cost, low scores are better). The fittest individuals are selected to be included in the next generation while the

least fit individuals are discarded. Individuals are then subjected to operations such as mutation and crossover which alter their genes. Mutation is the process of randomly altering one or more of the genes, it allows for exploration and prevents solutions from becoming stuck in local optima. Crossover is the process of combining the genes of two or more parents to create an offspring. The genes of fit parents are spliced together to hopefully create fitter individuals. Often, the concepts of crossover and mutation are combined into a single genetic operator. This process continues in a loop, generating populations which become fitter over time.

There are different types of EA including Genetic Programming (GP) - where solutions are in the form of computer programs - Genetic Algorithms (GA) - where solutions take the form of strings of numbers and crossover and mutation are often used together - and Evolution Strategies (ES) - where solutions are represented as vectors of real numbers, mutation is the primary operator which perturbs the population and each gene is mutated according to an adaptive step size.

Evolutionary algorithms are useful when trying to find an approximate solution to a difficult problem (i.e. when it is too expensive to find the exact optimal solution). They are good at exploring optimization problems as they are random in nature, fairly robust to noise and do not make any assumptions about the fitness landscape. [23, p. 104-105]. Mutation allows for random exploration of the search space, crossover allows for useful information to be shared between individuals and the fitness function makes sure that this random exploration of the search space is goal-directed.

2.3.1 Differential Evolution

Differential Evolution (DE) is a commonly used EA [24]. It is explainable, simple to implement and use, has few tunable control parameters and is easily parallelisable. In DE, the population is randomly intialised, individuals in the population move to a new solution in the search space by using rules which combine the genes of other individuals. More specifically, individuals are altered according to the scaled differences between other randomly selected individuals. At every generation, each individual solution is randomly perturbed in this way, if the new solution is better than the original, it is used in the next generation, otherwise the original solution is used.

A simple variant of DE (as proposed in the by the original creators [24]) is illustrated in Algorithm 1.

Algorithm 1 Differential Evolution

1: Randomly initialize each individual $x_i \in \mathbb{R}^n$
2: **while** termination criterion not met **do**
3: **for** each individual x in population **do**
4: Randomly select three individuals a, b, c from population which are distinct from each other and distinct from x
5: Randomly select $R \in \{1, 2, ..., n\}$
6: $y \leftarrow x$
7: **for** each dimension $d = 1, 2, ..., n$ **do**
8: Generate random number $r_d \sim U(0, 1)$
9: **if** $r_d < CR$ or $d = R$ **then**
10: $y_d \leftarrow a_d + F(b_d - c_d)$
11: **if** $cost(y) \leq cost(x)$ **then**
12: $x \leftarrow y$

The crossover probability parameter (CR) determines what proportion of an individuals genes are updated. Line 10 shows how an individual is perturbed according to a base vector (a) and the difference of two other vectors (b and c). The severity of this perturbation process is controlled by the differential weight

parameter (F).

DE is a versatile and competitive evolutionary algorithm [25]. DE has been applied to solve many problems including robot path planning [26] and power system scheduling problems [27].

2.3.2 Covariance Matrix Adaptation Evolution Strategy

One of the most popular and successful EAs is Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [28]. In CMA-ES, candidate solutions are sampled from a multivariate normal distribution which represents a statistical model of the population. The worst performing solutions are discardedd and the remaining solutions are used to update the distribution parameters.

CMA-ES is one of the most powerful optimization algorithms. CMA and its variants often win the Black Box Optimization Benchmarking competition [29].

Algorithm 2 Covariance Matrix Adaptation Evolution Strategy

1: Initialize $C = I, p_c = 0, p_\sigma = 0, \sigma = \sigma_0$, $m = m_0$
2: **while** termination criterion not met **do**
3: Sample solutions: $x_i = m + \sigma y_i$, where $y_i \sim \mathcal{N}_i(0,\, C)$ for $i = 1, ..., \lambda$
4: Update mean: $m \leftarrow m + c_m \sigma y_w$, where $y_w = \Sigma_{i=1}^{\mu} w_{rk(i)} y_i$
5: Update σ path: $p_\sigma \leftarrow (1 - c_\sigma)p_\sigma + \sqrt{1 - (1 - c_\sigma)^2}\sqrt{\mu_w} C^{-\frac{1}{2}} y_w$
6: Update C path: $p_c \leftarrow (1 - c_c)p_c + 1_{[0,2n]}\{\|p_\sigma\|^2\}\sqrt{1 - (1 - c_c)^2}\sqrt{\mu_w} y_w$
7: Update σ: $\sigma \leftarrow \sigma \times \exp\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|p_\sigma\|}{\mathbb{E}\|\mathcal{N}(0, I)\|} - 1\right)\right)$
8: Update C: $C \leftarrow C + c_\mu \Sigma_{i=1}^{\lambda} w_{rk(i)}(y_i y_i^T - C) + c_1(p_c p_c^T - C)$

Algorithm 2 demonstrates how the CMA-ES algorithm operates. This pseudocode is taken from [30]. λ is the population size, y represents the new solutions drawn from the distribution. Line 2 shows how new candidate solutions are drawn from the distribution. Lines 4-8 update the parameters which control the position and shape of the distribution. On line 4 the distribution mean m (position of the distribution in the search space) is updated according to weighted sum (controlled

by the ranked weights w) of the best solutions in the current generation (solutions with a better cost have a larger impact on how the mean is updated). Line 7 updates the step size σ which controls how aggressively the mean is moved each generation. Line 8 updates the covariance matrix C which controls the shape of the distribution. Both σ and C are updated according to evolution paths p_σ and p_c respectively. These evolution paths record a moving average of previous updates and give momentum to the C and σ update process. c_m, c_σ and c_c control how aggressively the mean, σ evolution path and C evolution path are updated respectively.

Hansen proposed a method of injecting external solutions into CMA-ES [31]. External solutions are added to the population during the evolution process. The CMA-ES algorithm is altered to make it robust to external solutions. If external solutions are not drawn directly from the CMA-ES population distribution, the update steps need to be tightly normalized in order to avoid a severe degradation of the algorithms performance. Injecting external candidate solutions in this way can lead to improved convergence speed.

2.4 Particle Swarm Optimization

Particle Swarm Optimization (PSO) [32] is a swarm intelligence heuristic search algorithm inspired by how birds and fish flock and move to form formations [32]. PSO works in a similar way to an EA in that global/societal information is used to perturb solutions with the goal of reaching an optimal solution. Each candidate solution is represented as a particle moving around the solution space. Each iteration, the particles position is updated according to its velocity. It's velocity is updated according to a simple mathematical rule. The velocity of a particle is influenced by the best position seen by that particle and also the

best position seen by any particle in the population. PSO differs from other evolutionary metaheuristics in that PSO does not operate directly on candidate solutions but instead on the velocity of candidate solutions. In this way, particles may fly past local optima (in a similar way to gradient descent with momentum).

Algorithm 3 Particle Swarm Optimization

1: **for** each particle $i = 1, 2, ..., popsize$ **do**
2: Randomly initialize particle position x_i
3: Randomly initialize particle velocity v_i
4: Initialize particles personal best position to current position $p_i \leftarrow x_i$
5: **if** $cost(x_i) < cost(g)$ **then**
6: Update global best position $g \leftarrow x_i$
7: **while** termination criterion not met **do**
8: **for** each particle $i = 1, 2, ..., population\ size$ **do**
9: **for** each dimension $d = 1, 2, ..., N$ **do**
10: Generate random numbers $r_p, r_g \sim U(0, 1)$
11: Update particle velocity $v_{i,d} \leftarrow wv_{i,d} + C_1 r_p(p_{i,d} - x_{i,d}) + C_2 r_g(g_d - x_{i,d})$
12: Update particle position $x_i \leftarrow x_i + v_i$
13: **if** $cost(x_i) < cost(p_i)$ **then**
14: Update particle's personal best position $p_i \leftarrow x_i$
15: **if** $cost(p_i) < cost(g)$ **then**
16: Update global best position $g \leftarrow p_i$

Algorithm 3 describes how PSO operates. The core workings of the algorithm is the process of perturbing solutions which is shown on lines 10 and 11. On line 11, the particles velocity is updated according to a weighted sum of **I**: the particles current velocity vector (controlled by the momentum parameter w), **II**: a vector which will pull the particle towards the global best seen solution (controlled by the local influence parameter C_1) and **III**: a vector which will pull the particle towards its personal best seen solution (controlled by the global influence parameter C_2). This velocity update step is stochastic (on account of the randomly generated numbers r_p and r_g). On line 12 the particles position is updated according to its velocity.

PSO and its variants are applicable to tasks in many areas including automatic control systems and electronic/electrical as well as civil and mechanical engineering [33].

DE, PSO and CMA-ES are all popular evolutionary algorithms for a number of reasons. DE and PSO are very easy to understand and implement and have a small number of tuneable parameters. The CMA-ES algorithm (as illustrated in Algorithm 2) is somewhat esoteric and is more difficult to understand and implement compared to PSO and DE. However, CMA-ES is one of the most successful evolutionary algorithms.

Since their original publications there have been many variants of DE [25], PSO [34] and CMA-ES [35, 36, 37], many of which are tailored to more specific use cases.

2.5 Neuroevolution

Neuroevolution uses EAs to evolve neural networks. Many neuroevolution approaches use EAs to evolve the weights of neural networks with fixed topologies. Each gene in the genotype corresponds to the value for a specific weight parameter in the NN.

As mentioned in Section 1, neuroevolution has been shown to be a competitive alternative for RL [12, 11]. Salimans et al. use a type of ES called a natural evolution strategy (NES) to evolve the weights of deep NNs [12]. Networks are evolved to play Atari 2600 games and to solve continuous robotic control tasks at a competitive level compared to Deep Q-Learning [38] and policy gradient methods such as A3C [39].

The NES used by salimans et al. [12] can be interpreted as a gradient-based method. It estimates a gradient in the search space and takes small steps in

that direction (not unlike backpropagation gradient descent). Such et al. use a simple gradient-free evolutionary algorithm to evolve very large NNs to play Atari games, outperforming DQN and A3C on some games [11]. Mason and Grijalva show that Neuroevolution is a competitive alternative to RL methods for control of heating, ventilation and air-conditioning (HVAC) systems [40]. These results confirm that simple EAs with modern computational resources can be used to efficiently train deep NNs for RL problems.

Neuroevolution offers attractive advantages over deep learning when it comes to solving RL problems [12]. Neuroeovolution does not suffer from many of the problems that plague DRL. Neuroevolution algorithms are indifferent to the nature of rewards (dense or sparse). They are tolerant of arbitrarily long time horizons and do not suffer from the credit assignment problem (determining which actions were most influential in reaching the reward). There is no need to calculate and backpropagate the gradients and so the fitness (loss) function does not need to be continuous or differentiable, indeed the activation functions of the NN itself do not need to be differentiable. When training Deep NNs with backpropagation, an architecture needs to be chosen beforehand. Extensive experimentation and human expertise/experience are needed in order to choose an appropriate architecture. Modern Neuroevolution techniques can intelligently evolve the network architecture to suit the task [41]. For the reasons discussed above, neuroevolution is a method which is particularly suited to tackling the task of robotic arm control.

DE is a natural first choice when evolving NN weights. The update process is somewhat analogous to the way the weights are updated using gradient descent (DE estimates the gradient as the difference between two random candidate solutions).

DE has been used before in neuroevolution in a variety of contexts [42, 43, 44,

45, 46, 47, 48]. Donate et al use DE to evolve the network topology and gradient descent to optimize the weights [42]. Wang et al. use a variant of DE to set initial network weights to avoid local minima when training with backpropagation gradient descent [43]. Baioletti et al. use DE to evolve the weights of networks to classify several datasets including MNIST [49], achieving accuracy comparable to state of the art approaches [45].

In the original publication of the PSO algorithm, the authors successfully apply PSO to optimize the weights of simple neural networks [32] (these networks are relatively shallow and solve simple problems such as the X-OR problem and classifying the Fisher Iris dataset [50]). Since then, there have been many approaches which use PSO and its variants to train the weights [51, 52] and to train the weights and architectures [53] of deep neural networks. It has also been shown that using EAs such as PSO can converge faster than backpropagation for optimizing NN weights [52].

Espinal et al. compare the performance of PSO and DE on evolving the weights of shallow NNs trained on several well known classification datasets [54]. The results show that DE has better computational performance (far fewer operations required during training) but that PSO exhibits better classification performance.

CMA-ES has been shown to be an effective method of optimizing the weights of neural networks for RL tasks [55, 47, 56]. Igel and Christian use CMA-ES to evolve NNs to solve several variants of the pole-balancing problem [55]. Mason et al. use CMA-ES to train NNs to predict power demand [56].

CMA-ES, DE and PSO are atractive EAs for evolving NNs to solve robot control problems, as evidenced by the results discussed above.

2.6 Neuroevolution of Network Topology and Weights

The neuroevolution techniques discussed so far operate by evolving the weight values of a neural network with a fixed topology. With these techniques, a topology must be decided upon by the user. It can be a difficult and arduous task to determine a suitable architecture for a given problem. There exists a set of neuroevolution algorithms which can evolve not only the weights of the network but also the topology of the network.

2.6.1 Neuroevolution of Augmenting Topologies

Neuroevolution of Augmenting Topologies (NEAT) is one such algorithm [57]. NEAT uses a genetic algorithm with a direct encoding. That is, the genotype maps directly to the phenotype to evolve both the weights and the topology of the network simultaneously.

The initial population consists of very simple networks. By mutation, weight values are altered, new nodes and connections are added and connections are removed. When a previously unseen connection is added, it is given a unique innovation number. Using these innovation numbers, NEAT is able to identify connections and structures which have the same origin (which therefore should serve the same purpose) across different networks. This allows for crossover between networks of dissimilar topologies.

NEAT uses speciation to preserve structural innovation within the population. The candidate solution networks are grouped into species based on their "genetic distance". The genetic distance between two individuals is a sum of the number of genes (connections) which are particular to only one individual. The fittest individuals from each species are then selected for the next generation. Adding

a new component to a network architecture (adding a new connection with a random weight) will initially hurt its fitness. It will take time for the weights to evolve and let the architecture show its promise. Speciation allows for innovation and exploration in network topology.

Complexification is baked into the way that NEAT works. Initially, the population consists of simple networks. The weights of these networks are evolved and optimized. Simultaneously, the network topologies are slowly made more and more complex. NEAT produces networks which are capable of solving tasks with a minimal network size. In other words, the hypothesis language (ie. the NN architecture) grows to fit the task.

The original authors becnhmark NEAT on the cart pole problem and achieve state of the art performance [57]. NEAT has been used to tackle several robotic control tasks [16, 14, 13]. D'Silva and Miikkulainen use NEAT to create obstacle avoidance behaviour in a robotic arm [16].

2.6.2 Hyper-NEAT

Hypercube-based Neuroevolution of Augmenting Topologies (Hyper-NEAT) is an extension of the NEAT algorithm which uses an indirect encoding - specifically, a Connective Compositional Pattern Producing Network (CPPN) is used - to evolve large neural networks [58].

A CPPN is simply a variation of a neural network in which different activation functions can be used at each node [59]. The phenotype network nodes are laid out in a substrate (geometric ordering of nodes). Nodes can be laid out in different substrates to suit the task at hand. The coordinates of a pair of nodes is fed into the CPPN. The CPPN outputs the weight value for the connection between that pair of nodes. Whereas NEAT directly evolves the architecture and weights of networks, Hyper-NEAT evolves a population of CPPNs. The CPPNs

then "paint" weight values onto the geometry of the substrate to produce the phenotype network.

By using a substrate and CPPNs, HyperNEAT is able to produce networks with spatial and structural symmetries, regularities and motifs. HyperNEAT enables a low-dimensional genotype to competently create high-dimensional networks. This indirect encoding enables Hyper-NEAT to scale easily to produce arbitrarily large neural networks.

Hyper-NEAT has been used for robotic control [60, 61, 62, 63]. Clune et al. use Hyper-NEAT to evolve gaits for quadruped robots [60]. Hyper-NEATs patterned reuse of phenotype modules is particularly suited to the symmetric problem of developing a multi-legged gait. Woolley and Stanley take advantage of Hyper-NEATs indrect encoding to evolve controllers for robotic arms with many segments [61].

2.7 Learning from Demonstration

Learning from Demonstration (LfD) and Imitation Learning refer to the paradigm of ML techniques that learn to mimic/imitate an expert [64]. Learning from demonstration can be set up such that it uses a supervised learning approach to solve a RL problem. Expert demonstrations of successfully completing the task are supplied to form a training set (collection of observation-action pairs for each time step of the task). An agent is trained using this dataset to learn a mapping from observations to actions. LfD is often used in learning robotic controllers [65, 66].

Karpov explores several methods of allowing humans to assist in the evolution of NN controllers to perform robotic control tasks in the NERO video game [67]. One such method uses human demonstrations to correct controller behaviour.

If evolved controllers deviate too much from human examples, the network is trained with backpropagation to imitate these examples. This neuroevolution with LfD method outperforms the base algorithm (neuroevolution without LfD) on most of the NERO tasks.

2.8 Robotic Arm Controllers

A traditional controller for a robotic arm would involve solving the inverse kinematic (IK) equations of the arm. Given the desired position and orientation of the end effector (gripper attached at the end of the robotic arm), a set of trigonometric equations can be solved analytically to determine the joint configurations/angles necessary to place the end effector in the desired position [68, p. 85-98]. One drawback of this approach is that IK controllers do not take into account the possibility of collision with obstacles making then unsuitable for dynamic environments.

Path planning algorithms can be used to search for a path that the robotic arm can take to reach a target [68, p 149-170]. These algorithms are computationally expensive since they have to search over a number of possible paths. Path planning and IK controllers are unsuitable for dynamic environments. If the starting joint positions or target positions change, then the problem needs to be solved all over again (the IK equations need to be solved again or the path planning algorithm needs to be run again).

2.8.1 Learning Approaches to Robotic Arm Control

As mentioned, DRL is commonly used for robotic arm control [7, 8, 9]. Amarjyoti highlights the successful applications of DRL to robotic control but also discusses some of the drawbacks of such approaches, namely the credit assignment problem

[8].

Gu et al. use a variant of Deep Q-Learning (DQL) (a DRL algorithm) to train a controller for a robotic arm [9]. DQL is used to train networks for three separate tasks: **I**: random target reaching (move the end effector of the arm to a randomly chosen position), **II**: pick and place (pick up object, move to desired location, place object) **III**: door pushing and pulling. These control tasks can cause issues for many RL algorithms due to the high sample complexity of the task caused by sparse and delayed rewards, something that Gu et al. [9] try to solve.

Evolutionary algorithms have long been used in the field of robotics. Hornby et al. evolve a set of parameters which are used to control the locomotion of an AIBO robot [69]. Lipson and Pollack evolve the bodies of robots which can locomote in the real world [70]. These approaches were early indicators of the success of evolutionary algorithms in the field of robotics.

There have been several successful applications of neuroevolution to robotic arm control tasks [13, 14, 15, 16].

Wen et al. evolve a NN controller for a planar musculor-skeletal arm (human-like robotic arm) [13]. NEAT is used to evolve two separate NN controllers to perform random target reaching (one which outputs the change in joint angles needed to reach the target and another which outputs the activation of each human-like muscle required to move the arm to the target location). Controllers are evaluated according to a pre-computed training set of joint configurations.

Huang et al. use limited human supervision to evolve a NN controller which can grasp objects of various shapes with a human-like robotic hand [14]. NEAT is used to evolve a controller which takes depth-sensor data as input and outputs the joint configurations necessary to grasp the object.

Moriarty uses neuroevolution to evolve a controller that can reach a target

whilst avoiding obstacles [15]. A neuroevolution algorithm called Hierarchical SANE is used to evolve two neural networks [71]. A primary network is used to move the end effector roughly close to the target while avoiding the obstacle and a secondary network is used to make more precise movements to the target position. Combining these networks produces a robust control policy which integrates both obstacle avoidance (primary network) and target reaching (secondary network).

The work most similar to this project is the work of D'Silva and Miikkulainen [16] which builds upon the work of Moriarty and Miikkulainen [15]. In this paper, the authors evolve NN controllers to move the end effector of a robotic arm to a target position in three different environments: **I**: environments with no obstacles, **II**: environments with stationary obstacles, **III**: environments with moving obstacles. In each of the three scenarios, the NEAT algorithm is used to evolve NN controllers. Sensory information from the environment (joint positions, position of target, position of obstacle) was used as input to the network. The network output determines how much to move each joint during each time step of the episode. The fitness/cost of each candidate solution is determined by using the controller in a simulated robotic arm environment and calculating how far away the end effector is from the target location (after the arm has been moved). Environment I can be handled by an inverse kinematics controller and environment II can be handled by a path finding algorithm. However, by using neuroevolution, a controller can be evolved which is robust to a dynamic environment with moving obstacles (an IK controller can't deal with obstacles, a path finding controller can't deal with moving obstacles, neuroevolution can deal with both). These results are evidence to the power of neuroevolution algorithms for creating robotic arm controllers.

This work will build upon previous work in using neuroevolution to control a robotic arm. While most approaches use sophisticated neuroevolution algorithms

such as NEAT to evolve complex network, this work focuses on using simple EAs to evolve the weights of simple NNs which can accurately control a robot arm. Furthermore, this work compares and contrasts different EAs and different approaches in an attempt to find the best system for evolving a NN controller. Finally, this work makes use of expert demonstrations to improve the evolution process, something which has been done before.

Chapter 3

Methodology

3.1 Robotic Arm

In order to evolve neural network controllers for a robotic arm, a robotic simulation environment is needed to evaluate the fitness of candidate solutions. CoppeliaSim (formerly V-REP) is a general-purpose robot simulation framework designed with algorithm development in mind [72]. CoppeliaSim provides a "ready-made" environment in which the robotic arm can "live". The framework accurately simulates the physics of robotics. CoppeliaSim provides a near real-world robotic environment.

A robotic arm was designed and assembled in CoppeliaSim. As can be seen in Figure 3.1 the arm consists of six links (excluding the base), connected by six rotational joints giving the arm six degrees of freedom (DOF). That is, the arm has six independent modes of movement. Joint 1 connects the base to link 1 (in this case the base is referred to as the parent link of joint 1 and link 1 as the child link), joint 2 connects link 1 to link 2 and so on. Joints 2, 3 and 5 rotate about a horizontal axis (with reference to their parent links) controlling the pitch between the parent and child links. Joints 1, 4 and 6 rotate about a

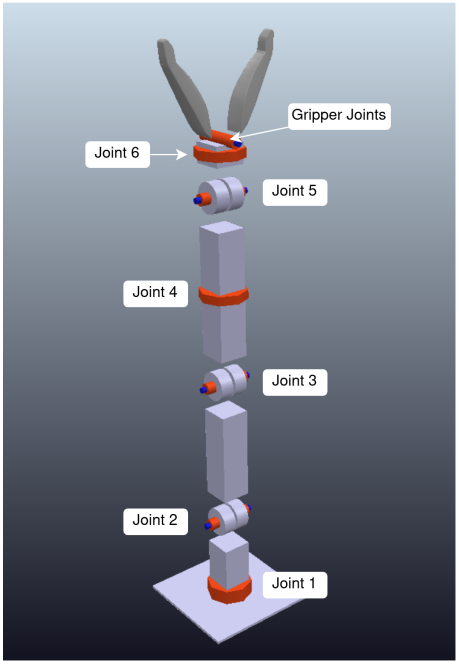

Figure 3.1: Robotic arm constructed in CoppeliaSim

vertical axis (with reference to the parent link) controlling the roll between the parent and child link. An end effector (gripper) is attached at the tip of the arm (joint 6). The gripper consists of two links which act as "fingers". The finger joints are connected to link 6 with two rotational joints giving the robotic arm a total of eight degrees of freedom when combined with the end effector. Joints 1, 4 and 6 have a minimum angle of -90 degrees and a maximum angle of 90 degrees (can rotate 90 degrees to the left and 90 degrees to the right). Joints 2 and 3 have a minimum angle of 0 degrees and a maximimum angle of 120 degrees (can pitch down 120 degrees). Joint 5 has a minimum angle of -120 degrees and a maximum of 120 degrees (can pitch up 120 degrees and pitch down 120 degrees). The finger joints have a movement range of 90 degrees (can position the fingers to be anywhere from perpendicular to or parallel to the final link of the arm). The arm is roughly 80 cm from base to end effector.

Although CoppeliaSim can be used to accurately simulate a robotic environ-

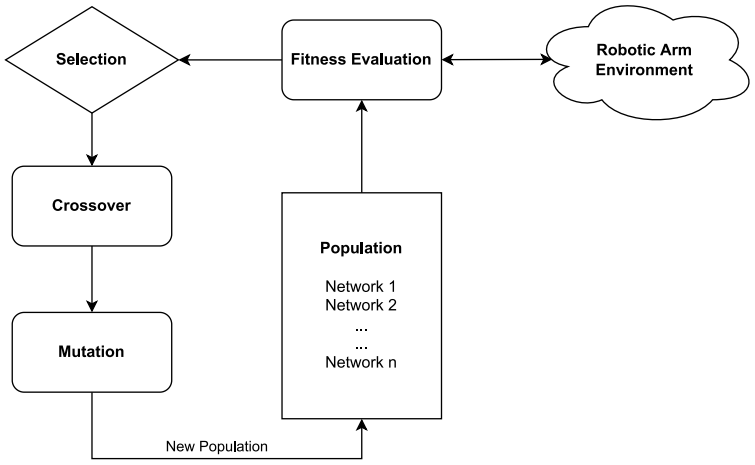

Figure 3.2: Schematic of neuroevolution working with robotic arm simulation.

ment, the simulations are too slow to be used for extensive experimentation with neuroevolution (thousands of simulations may need to be run for a single generation of a EA). PyRep is a tailored framework built on top of V-REP [73]. It is specifically designed for the purposes of rapid prototyping of robot learning algorithms. It provides a python API for robot control and offers a considerable simulation speed boost over the original CoppeliaSim python API. The NN controllers interact with the joints in CoppeliaSim's passive joint mode. This means that the robotic arms movements do not take the physics of the environment into account, thus making the learning tasks more approachable. If physics such as gravity and momentum were enabled, NN controllers would not only need to learn the correct joint positions but also how to move the joints in order to compensate for how the arm moves and responds in a physics environment. Furthermore, configuring the arm model and CoppeliaSim's physics engines in order to behave realistically would require a substantial amount of arduous and painstaking work.

The pyrep environment is integrated into the objective function of the EA as visualized in Figure 3.2.

3.2 Robotic Control Tasks

Work and experimentation in this project are carried out with the overarching goal of evolving a controller which can competently tackle the pick and place task. The goal of this task can be broken down into four steps or sub-tasks:

1. move the end effector to the object
2. grasp the object
3. move the object to desired the location
4. place the object.

NN controllers are evolved with the goal of placing the end effector as close as possible to random target positions. Controlling the movement of the arm makes up the bulk of the pick-and-place task (sub-task 1 and sub-task 3). Sub-tasks 2 and 4 are relatively easy tasks which can be performed using a simple program/routine and there is little motivation to apply a learning agent to perform these tasks.

The task of moving the end effector to a desired position was approached in three different ways:

1. NN ouputs the desired final joint positions needed to reach the target in a single action.
2. NN slowly changes the joint angles using multiple actions over several time-steps.
3. NN learns to imitate expert demonstrations of how the task should be performed.

Most of the work and experiments in this report focus on controlling only the six main joints of the arm. The finger joints do not control where the end effector is positioned so they are ignored when training controllers.

The six arm joints are used in all three approaches to movement control. The six joints are independent and are not intrinsically redundant. Each joint has an effect on where the end effector is placed. However, it is the case that some joints are functionally redundant in the context of moving the tip to a desired position. For example, the robotic arm can reach all required target positions without the use of joints 4 or 6. The reachable workspace of the arm (the points that the end effector can reach) using all six joints is the same as the reachable workspace while using only joints 1, 2, 3 and 5. However, joints 4 and 6 increase the dexterous workspace of the arm. While not allowing the end effector to attain new positions, joints 4 and 6 allow the end effector to reach those points from different orientations. Six degrees of freedom is the minimum to achieve an arbitrary position and orientation in space [74, p. 85-86]. By giving the NNs control of more joints, they are allowed to be more creative in how they move the arm and are given more freedom to achieve innovative and efficient solutions.

3.2.1 Data

Each candidate solution network evolved during the evolution process needs to be evaluated against a cost/fitness function which accurately reflects how good that network is at moving the robotic arm to a random target location. The cost of a network for the movement task is equal to the euclidean distance between the arms tip (end effector) and the target after the NN controller has finished moving the arm.

$$cost = \sqrt{(target_x - tip_x)^2 + (target_y - tip_y)^2 + (target_z - tip_z)^2}$$

Controllers are evolved with the goal of being able to accurately place the end effector at a target objects placed anywhere within a region in front of the arm. Targets objects are placed on the floor only. While this presents a simpler task than say, placing targets anywhere within a spherical region around the robot, it represents a more realistic use-case for the robot.

As more target positions are added to the training dataset, the evolved NN controller is able to generalise much better to unseen target positions but the training times increase. The training dataset consists of fifty target positions which offers a good compromise between generalization and training time.

A simple method of creating a dataset would be to uniformly sample positions within the target space. However this results in certain regions of the target space being under-represented and other regions being over-represented. In order to increase the controllers ability to generalize to unseen target positions, a new, more evenly distributed dataset was created. The target space is split up into fifty regions of roughly equal area. A target position is uniformly sampled within each of these regions. As can be seen in Figure 3.3, these fifty target positions give a better representation of the target space while still preserving some randomness in how targets positions are selected. Preliminary experiments show that training on this dataset results in improved generalization to unseen target positions when compared to training on target positions sampled completely at random.

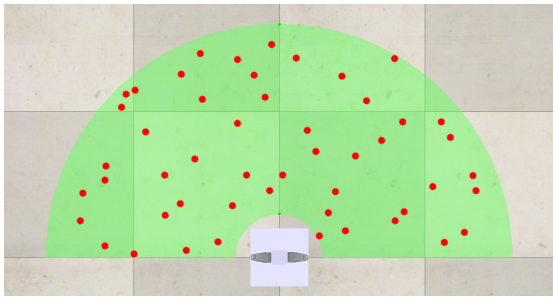

Figure 3.3: Visualisation of the dataset of fifty target positions (red dots) overlaid onto the target space (green region) of the robotic (as seen from above).

3.2.2 Single-Step Movement Approach

The goal of this approach is to evolve NNs which can place the end effector as close as possible to a target position. The NNs are provided with sensor inputs describing the state of the environment and they must output the joint angles/positions necessary to place the end effector at the target. The networks are evaluated using the set of fifty target positions described in section 3.2.1. The input consists of the x and y co-ordinates of the target. Because an episode consists of a single step (where the network outputs the desired joint position), there is no need to provide the joints positions as input to the network. The cost of a network is equal to the mean distance from the target over the dataset.

Conceptually, this approach is the simplest, giving the NN only the simplest state information and using it to perform only a single action.

An evaluation of a single network on a set of fifty target positions takes around 150 milliseconds. The main bottleneck of the evolution process is waiting for the CoppeliaSim simulation to complete.

3.2.3 Multi-Step Movement Task

The goal of the multi-step movement task is the same as that of the single-step movement task (to place the end effector at a target). The two methods are

differentiated in their method controlling the arm. This approach is similar to the one found in [16]. For this task, the NNs are provided with the following state information:

- Six joint angles that represent current joint state;
- x, y co-ordinates of target position;
- x, y, z co-ordinates of target position relative to the end effector.

In addition to the absolute position of the target, the network also receives the position of the target relative to the end effector of the arm at the current time-step (i.e. the position of the target from the point of view of the end effector). This additional state information tells the controller in what direction and how far it needs to move in order to reach the target.

Networks are evaluated according the same fifty positions as in the single-step task. For each time-step, the network receives the state information and outputs a delta for each joint, which determines how much the angle of each joint should change. The joint deltas are normalized to be in the range [-5, +5] degrees. The network has an additional output node which controls whether or not to stop the episode. The idea is to provide the network with an easy way to decide when it is finished moving instead of forcing the network to set all of the joint deltas to 0. The episode begins by placing the target and setting the joints to their starting positions. The episode ends if the value outputted by the stop node is greater than 0.5 or after twenty time-steps are completed. The cost of a network is the mean distance to the target.

An evaluation of a single network on a dataset of fifty target positions takes around 500 milliseconds. Again, the main bottle-neck is the CoppeliaSim simulation.

3.2.4 Imitating Expert Demonstrations

This approach to controlling the arms movement involves imitation learning (learning from demonstration). The goal of this task is to evolve a network to imitate a set of expert examples which demonstrate how to move the arm in order to accurately complete the movement task.

The expert demonstrations are perfect examples of how to solve the single-step task. Despite the multi-step approach outperforming the single-step approach (as evidenced by the results in section 5.2, the single-step movement approach is more suited to imitation learning. Not only would it be more difficult to generate expert demonstrations for the multi-step approach, forcing the multi-step network to imitate expert demonstrations would limit its creativity and freedom. For any given target position, a single-step control network can reach the target in only a handful of ways (there are only a few joint configurations which would result in the end effector being placed at the target). A multi-step network, however, has much more freedom in how it reaches the target as it chains together actions over multiple time steps.

Each entry in the expert training dataset consists of the x, y co-ordinates of the target positions (the same target positions found in the training set for the single-step and multi-step movement tasks are used) along with the joint angles necessary to place the end effector at these targets. For each target position, IK are used to correctly position the end effector at the target. In order to control the arm with IK, both the target positions and the target orientation are needed. The target orientation is initially set so that the end effector "attacks" the target from directly above. If no IK solution is possible for this orientation (i.e. there is no joint configuration which results in the end effector being placed at the target in the given orientation), the pitch of the orientation is progressively altered until an IK solution is found. This is done to ensure that the set of expert

demonstrations are in some way continuous, allowing the NN to better represent them. When the target is placed on the ground, the end effector pose (position and orientation) is too far away from the target pose for the IK equations to be solved directly. Instead, a random search over joint configurations (positions) is carried out until a configuration is found which results in the end effector pose being close enough to the desired pose to solve the IK equations. This process highlights a disadvantage of using IK to control the arms movement. Finding IK solutions becomes a slow process when the end effector is far away from the target (as is the case for this work) because random sampling has to be used.

The dataset consists of fifty expert demonstrations. The expert demonstrations are automatically generated so it is possible to create a larger dataset. However, the number of expert demonstrations is kept low as a consideration of other real-world robotics tasks. If this approach was to be applied to a more complex robotic arm control task, expert demonstrations may need to be created by hand.

Networks are evaluated on the fifty expert demonstrations. The x, y coordinates of the object are fed into the network. The network outputs the joint angles. The cost is equal to the mean squared error between the output joint angles and the expert joint angles.

The imitation of expert demonstrations can be viewed as a supervised learning approach to the single-step movement task. A RL approach to the movement task bypasses the need for collecting/generating training data and also allows for innovation in how NN controllers approach the task. Using an imitation learning approach restricts the creativity of solutions but could provide for an easier learning/optimization problem. In the context of solving the movement task, imitation learning does not require networks to be evaluated in the simulation environment. This means the training times for the imitation approach are drastically reduced

(for the imitation task, a single evaluation takes around 2 ms compared to 150 ms for the single-step task).

The final approach to the movement task involves combining the imitation approach with the single-step approach. Networks evolved on the imitation task are used as seed networks for the single-step task. The imitation network is used as a starting point for evolution/training on the single-step task. In the context of CMA-ES, the initial mean value (the mean can be viewed as the algorithms current "favourite" solution) is set to be equal to the weights of the imitation network. Furthermore, the seed solution is injected into the population until a solution with a lower cost is found. This is done to help "shape" the distribution around the seed solution (update the parameters that control the distribution so that CMA-ES will sample solutions around the seed). The seed network is essentially pretrained on the imitation task and fine-tuned on the single-step task. A network that performs well on the imitation task will likely perform well on the single step task. Furthermore, training is significantly quicker on the imitation task so a good starting point for the single-step task can be found relatively quickly.

3.3 Supervisor Network

The evolved movement control networks are used to perform the pick-and-place task by way of a supervisor network. The supervisor network is evolved to predict which sub-task is to be performed at each step of the pick-and-place task. To complete the pick-and-place task, each sub-task is performed by a sub-routine. Sub-tasks 1 and 3 (moving to object and moving to target) are performed by a movement control network (using either the single-step or multi-step approach). Sub-tasks 2 and 4 (grasping object and releasing object) are performed by non-

learning programs. Sub-task 2 is completed by simply rotating the finger joints until the fingers come into contact with and grip the object. Sub-task 4 is performed by simply moving the finger joints to their initial positions, thus releasing the object.

A NN controller is constructed which can call four sub-routines to perform each of the four steps of the pick-and-place task. The supervisor network has a similar architecture to the networks used for movement control. That is, the network has a single hidden layer with twenty nodes. At the start of an episode, the arm joints are set to their starting positions and the object and target are placed randomly on the floor in front of the arm. Throughout the task, the supervisor network is given four opportunities to select which sub-task to perform. At each of those opportunities the state of the environment is fed into the network and the network output determines which sub-task to perform (the output layer consists of four nodes and uses a softmax activation). After the sub-routine is complete and the arm has moved, the supervisor network is prompted to select the next sub-routine to call and so on until each of the four sub-routines have been called and the task is complete. When prompted to select the appropriate sub-routine, the supervisor network is given the following state information:

- x, y co-ordinates of object
- x, y co-ordinates of target
- Eight joint angles (six arm joints and two finger joints)

The supervisor network is trained using a supervised learning approach. That is, the network is evolved/trained on a dataset of (state information, label) pairs gathered using the pick-and-place controller. The pick-and-place task is performed 5,000 times using a multi-step controller for sub-tasks 1 and 3 and a non-learning program for sub-tasks 2 and 4. When constructing the dataset,

each sub-task is called in the correct order. At each of the four steps, the state information (inputs) and the ID of the sub-task which should be performed (labels) are recorded. The dataset therefore consists of 20,000 entries (each of the four sub-tasks being completed 5,000 times). During evolution, the networks are evaluated against this dataset. The cost is equal to the mean squared error between the network outputs (vector of length four fed through softmax activation function) and the labels (one-hot encoded vector of length four).

When evolving a multi-step network, the joints are set to the same positions at the start of every episode. An undesirable consequence of this is that the evolved controller is only able to perform competently when the arm joints are set to the starting positions. In order to perform the pick-and-place task with a multi-step network, the arm needs to be reset to its initial position after subtask 2 (grasping object) and before subtask 3 (moving to target). If the arm has already been moved to the object, the multi-step network will not be able to move from that pose to a pose which moves the end effector to the target.

3.4 Neural Network Parameters

This work focuses on using using EAs to evolve the weight and bias parameters of a NN with a fixed topology. The NN architecture used for movement control has ReLU activations after the hidden layers and sigmoid activations after the output layer. Outputs for each node are in the range [0, 1], these outputs values are then mapped to ranges appropriate for the task. For the single-step task, sigmoid outputs are scaled to the range [minimum joint angle, maximum joint angle]. For the multi-step task, sigmoid outputs are scaled to the range [-5, 5].

This project focuses on CMA-ES, DE and PSO as the neuroevolution algorithms. DE and PSO are easy to program and to use and they have few tunable

hyperparameters. CMA-ES is one of the most powerful and popular EAs. The idea is to find simple, easy to understand and effective approaches to robotic arm control.

Chapter 4

Experimental Settings

Experiments were run using all three approaches to movement control of the robotic arm (single-step, multi-step, imitation). The training times of the single-step and the multi-step task meant that extensive search over EA hyperparameters was not feasible (100,000 evaluations on single-step and multi-step tasks takes around five hours and thirteen hours respectively). A hyperparameter search was carried out for each of the three algorithms (DE, PSO, CMA-ES) on the imitation task (100,000 evaluations takes around five minutes). A grid search over hyperparameters was performed for each of the three algorithms. The algorithms were run a total of five times for each combination of hyperparameters for 100,000 cost function evaluations. The hyperparameters which achieved the best mean cost over the five runs were selected as optimal.

The best hyperparameters found for DE are as follows:

- population size = 200
- differential weight = 0.2
- crossover probability = 0.9

The best hyperparameters found for PSO are as follows:

- population size = 100
- momentum = 0.4
- local influence = 1.5
- global influence = 2

CMA-ES has many more hyperparameters than PSO and DE. Most of these hyperparameters are carefully set by the creators [28]. The values of these hyperparameters depend on the dimension (number of decision variables) of the problem and should be robust to different objective functions. A grid search was performed over some of these hyperparameters, namely the initial step size σ_0 (initial step-size), population size, μ (number of solutions used to update distribution) and the initial value for m (distribution mean). It was found that σ_0 and the initial value for m had the biggest impact on performance. The best value for σ_0 was found to be 0.1. The best value for initial m was found to be 0 (the "origin" in the search space). Changing any of the other hyperparameters did not result in any significant improvement in performance.

The hyperparameters found for each algorithm on the imitation task are used for all experiments. While the hyperparameters are optimal (or near-optimal) for the imitation task, it is unlikely that they are optimal for the single-step or multi-step task. However, due to the similarities between the the tasks it is likely that the hyperparameters will result in good performance on the single-step and multi-step tasks. For each of the three tasks, the EAs are evolving the weights of networks with identical architectures. If a certain set of hyperparameters are optimal for evolving a network to perform one task, then it is not unreasonable to assume that those same hyperparameters will be good choices when evolving that same network to perform similar tasks. The performance of hyperparameters

which were found to be optimal for the imitation task were validated on the single-step and multi-step tasks.

All experiments except those discussed in sections 5.5 and 5.6 were run for 100,000 cost function evaluations. The best evolved network from each run was tested on a set of 1000 unseen target positions. Each experiment was repeated five times.

The cost values shown in the figures in chapter **??** typically range between the values 0 and 1. For context, these cost values (unless otherwise specified) represent the distance from the end effector to the target position in metres. If a network achieves a cost of 0.05, this means that the network is able to move the end effector to within 5cm of the target (averaged over all targets in the dataset).

The cost plots such as Figures 5.2 and 5.3 show the mean cost of each algorithm/approach over five runs. Furthermore, the minimum and maximum cost at each generation (over the five runs) is highlighted.

4.1 Experiments

Below is a list of all experiments and analyses carried out.

- Performance comparison of CMA-ES vs DE vs PSO on the movement task using the single-step approach (Section 5.1).
- Performance comparison of CMA-ES using the single-step approach vs the multi-step approach (Section 5.2).
- Evaluating the performance of CMA-ES on the imitation task (imitating expert examples which show how to correctly solve the single-step movement task, Section 5.3).

- Evaluating the performance of CMA-ES on the single-step task using expert demonstration seed networks (Section 5.3).
- Analysing the effect of disabled joints on the ability of NN controllers to move the robot arm.
- Evaluating the performance of CMA-ES on evolving a supervisor network which determines when to perform each of the pick-and-place sub-tasks.

Chapter 5

Results

5.1 Single-Step Task

This section compares the performance of DE, PSO and CMA-ES on the single-step movement task.

Figure 5.1 visualizes the spread of solutions found by each algorithm in terms of average distance to target on unseen target locations. Each boxplot consists of five observations each (one for each run). Table 5.1 shows the average distance to target of solutions evolved by each algorithm on the training set and test set.

Figure 5.1 clearly shows that CMA-ES performs much better than the other two algorithms. DE performs slightly better than PSO. Both CMA-ES and DE are more consistent in the performance of their best evolved networks. One-tailed Wilcoxon Rank-Sum tests with a significance level of 0.01 conclude that CMA-ES achieves significantly lower cost (distance to target) than both PSO and DE.

Figure 5.2 shows the cost (distance to target) of the best solution found at each generation for each of the three algorithms. It can be seen that each of the three algorithms are still finding better solutions right up to 100,000 evaluations. In other words, the algorithms have not fully converged. Running each algorithm

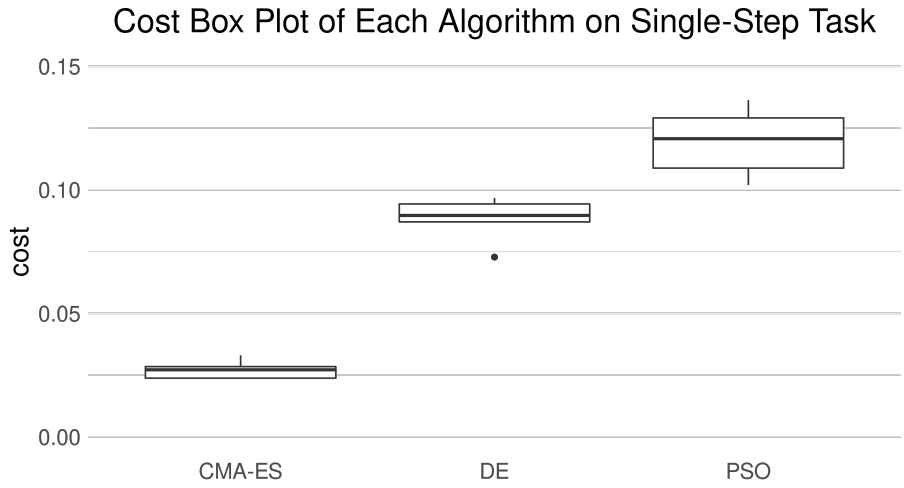

Figure 5.1: Boxplot of cost for DE, PSO, CMA unseen target locations

Algorithm	Mean Train Cost	Std. Train Cost	Mean Test Cost	Std. Test Cost
CMA-ES	0.0195	0.0037	0.0272	0.0039
DE	0.0822	0.0109	0.0881	0.0094
PSO	0.1042	0.0173	0.1194	0.0141

Table 5.1: Performance on single-step task averaged over five runs

for longer will likely result in better performing networks. It is clear, however, that CMA-ES is the superior algorithm for the single-step task. Aside from the fact that CMA-ES seems to be a good algorithm for neuroevolution, the choice of hyperparameters could partially explain why CMA-ES outperforms the other algorithms by a significant margin. When performing the hyperparameter search, CMA-ES was less sensitive to hyperparameters as compared to the other two algorithms. Furthermore, the best population size found for CMA-ES (19) was considerably lower than the population size for both PSO (100) and DE (200). CMA-ES performs well with a smaller population size and can therefore be run for more generations and will have "more time" to find better solutions within the constraints of an evaluation budget of 100,000.

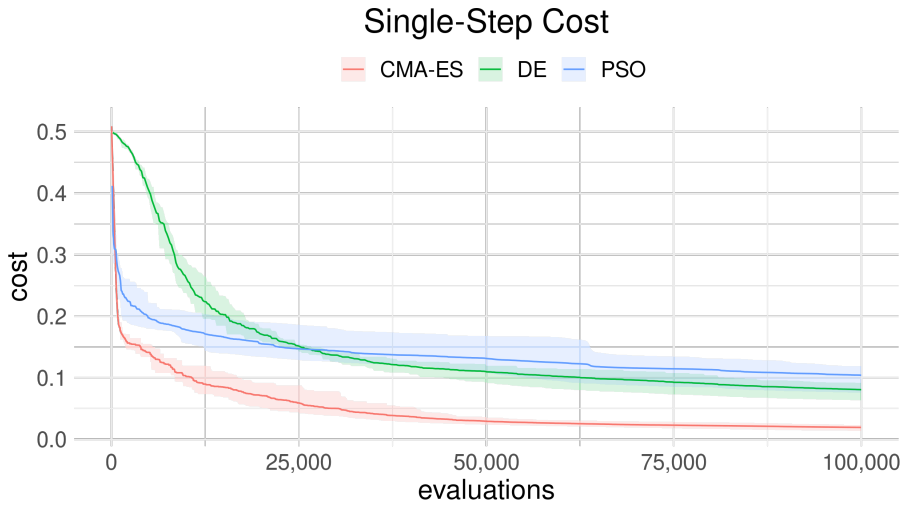

Figure 5.2: Cost of DE, PSO, CMA on single-step movement task

5.2 Mulit-Step Task

This section compares the performance of the single-step approach and the multi-step approach. Due to the superiority of CMA-ES on the single-step task, it was used as the EA for all remaining experiments. Experiments focus on using CMA-ES to evolve networks for both movement control approaches.

CMA-ES was used to evolve two different networks for the multi-step task. The first network receives the position of the target relative to the end effector as input while the second network does not.

Figure 5.5 shows the spread of test costs achieved by networks trained on the single-step task, the multi-step task without relative position information and the multi-step task with relative position information.

Without relative position information the multi-step network is not able to match the performance of the single-step network. In Table 5.2 it can be seen that using the multi-step network without relative position information results in the end effector being approximately 12cm further away from the target than the single-step network. However, if the multi-step network receives relative position

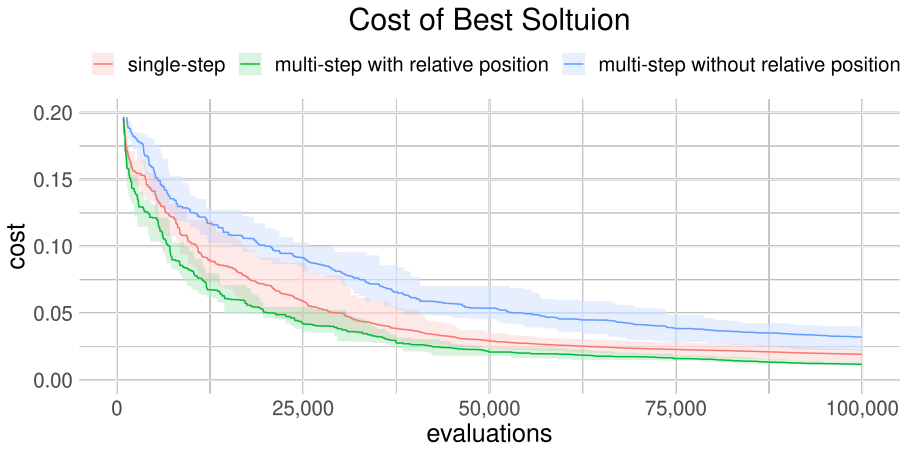

Figure 5.3: Cost of best solution found at each generation by CMA-ES on I: single-step task, II: multi-step task with relative position information and III: multi-step task without relative position information

Approach	Train Cost	Std. Train Cost	Test Cost	Std. Test Cost
Single-step	0.0195	0.0037	0.0272	0.0039
Multi-step (relative position)	0.0122	0.0016	0.0165	0.0021
Multi-step (no relative position)	0.0322	0.0067	0.0394	0.0056

Table 5.2: Performance of CMA-ES on single-step and multi-step approaches averaged over five runs

information it results in the end effector moving approximately 7cm closer to the target than the single-step network.

Figure 5.4 visualizes the ability of the single-step and multi-step networks to generalize to unseen target positions. For each approach (single-step and multi-step with relative position information), the train and test cost of five evolved networks is plotted. It can be seen that the multi-step networks not only achieve better training performance than the single-step networks but their performance does not degrade as much when evaluated on unseen target positions. A one-tailed Wilcoxon Rank-Sum test with a significance level of 0.01 concludes that the multi-

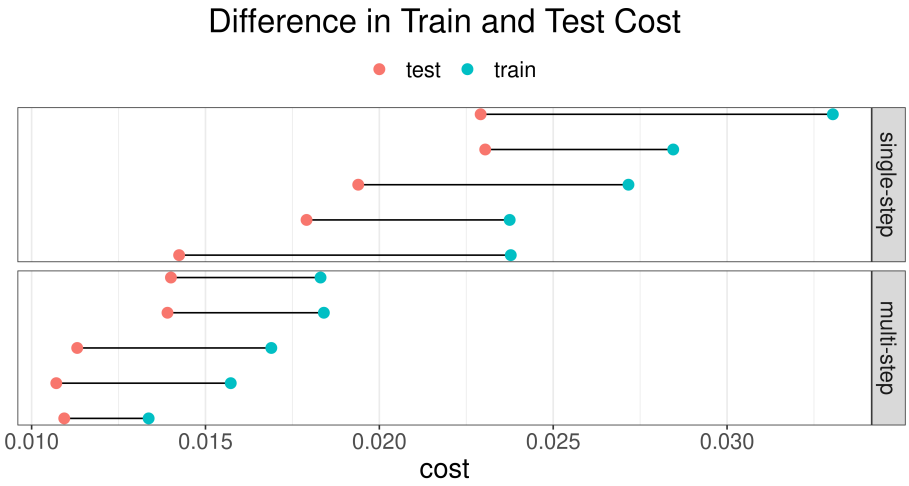

Figure 5.4: Comparison of generalization between single-step (top) and multi-step (bottom) approaches. Each line represents the difference between train and test cost for a single run.

step approach achieves significantly lower cost (distance to target) compared to the single-step approach.

In all evolved multi-step networks, the stop node is never activated. The network instead chooses to either set joint deltas to 0 degrees when it is finished or space out the joint deltas to fit into the twenty time-step window. In other words, the network controllers are fitted to the episode length that they are trained on. If an evolved controller is given fewer or more time-steps to move the arm, performance will worsen.

It may seem somewhat counter intuitive that multi-step outperforms single-step. The movement task can be completed with a single forward pass of the single-step network. To complete the same task with a multi-step network, twenty forward passes are needed. The multi-step network is tasked with chaining together several actions whereas the single-step network only performs a single action. Furthermore, the multi-step network accepts the joint positions as input and needs to learn how to effectively process and make use of them, increasing

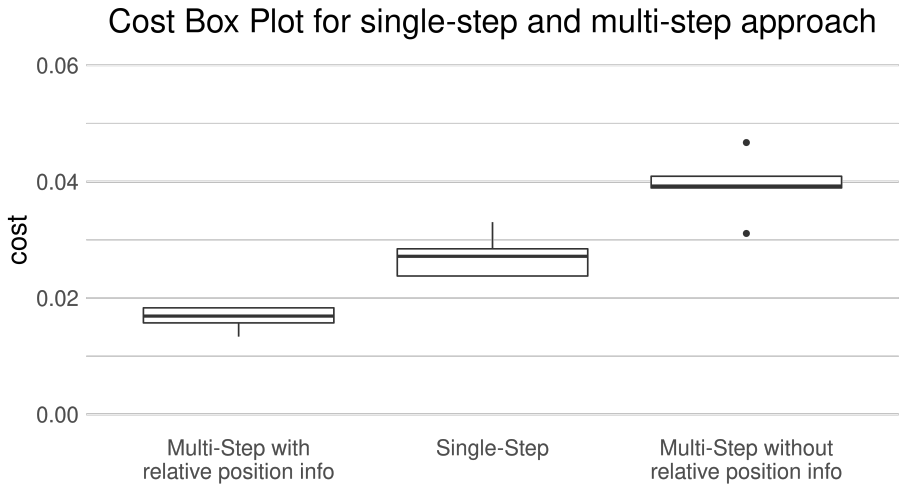

Figure 5.5: Cost of networks trained of multi-step and single-step networks

the number of network weight parameters and increasing the difficulty of the task from an optimization standpoint (the single-step network has 186 weight parameters while the multi-step network has 387).

Despite this, the multi-step network consistently outperforms the single-step network. This result poses an interesting trade-off between the two networks in terms of speed against accuracy. The single-step network can be trained in a third of the time it takes to train the multi-step network for the same number of objective function evaluations. The single-step network can also perform the movement task much quicker at test time than the multi-step network, albeit with less accuracy.

If relative position information is not given to the multi-step network, its performance degrades below that of the single-step network (although still performing fairly competitively). This indicates the usefulness of the relative position information and more generally, the need for practical state information. The position of the target relative to the end effector is in essence, an engineered version of the absolute position of the target. In theory the network could calculate/learn the relative position of the target from the other input features. However, the

network having only one hidden layer of twenty nodes means that feature engineering has a bigger impact on performance.

5.3 Learning from Expert Demonstrations

This section discusses the results of using expert demonstrations to evolve movement control networks.

Figure 5.6 shows the cost of CMA-ES trained on the imitation task. The imitation cost of the best network steadily decreases for the first 2,000 or so steps. However, these networks start to overfit quite early and fail to accurately imitate unseen expert demonstrations. The best network from each generation is evaluated on the single-step task. Despite the network failing to generalize to unseen expert demonstrations, the distance to the target on the single-step task continues to decrease.

Figure 5.7 shows that when the imitation approach and the single-step approach are run for 100,000 evaluations each, the imitation approach performs significantly worse. Even though both approaches evolve a network controller to move to the same fifty target positions, tackling the task from a RL approach (single-step network) results in better performance compared to a supervised learning approach (imitation network). It is likely, however, that with more training data and more generations, the performance of the imitation approach would improve.

Figure 5.8 shows the cost of CMA-ES on the single-step approach trained with and without a seed network. When a seed is used, the algorithm converges much quicker and the final performance is more reliable. Ultimately, the final distance to target is not significantly improved when a seed network is used. A two-tailed Wilcoxon Rank-Sum test with a significance level of 0.01 concludes that there is

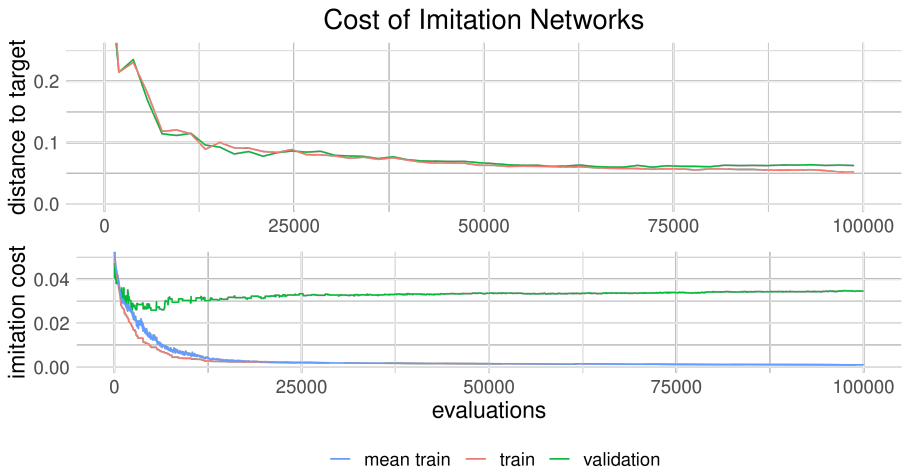

Figure 5.6: Convergence graph of cost networks evolved using the imitation approach. The best network from each generation is evaluated on the imitation task (Bottom) and on the single-step task (Top)

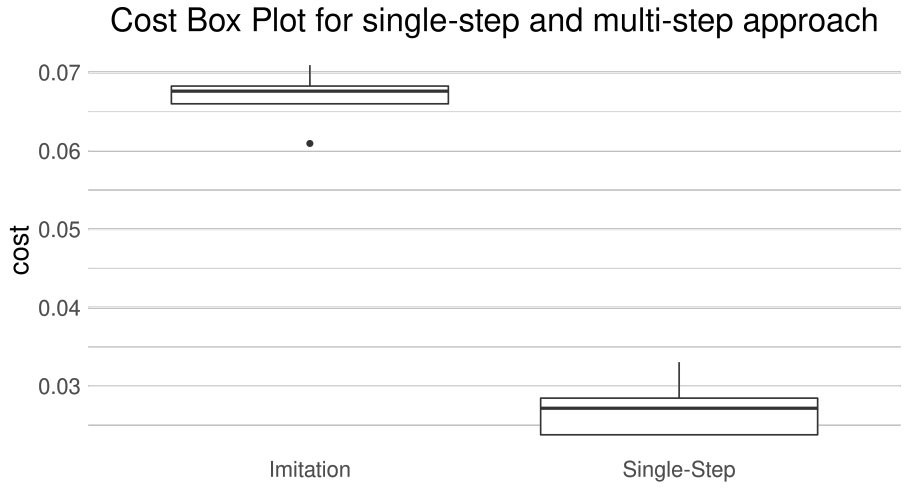

Figure 5.7: Learning from environment interactions vs learning from expert demonstrations

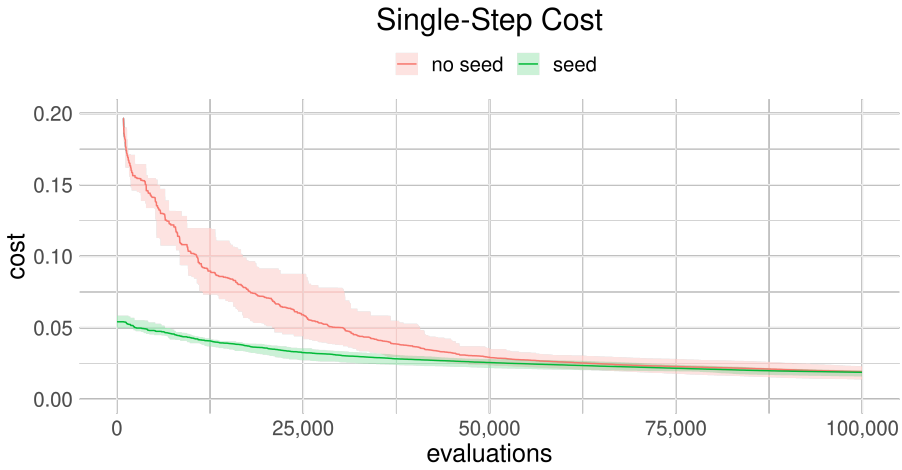

Figure 5.8: Convergence with and without expert demonstration seed network

no significant difference in the performance between using a seed network and not using a seed network. This approach is limited by the quality of the seed network.

5.4 Movement With Disabled Joints

When robot arms are deployed in real-world environments they can suffer damage which can impair their ability to perform as intended. A joint could be damaged, rendering it incapable of moving. A controller could be incapable of successfully completing its intended task if a joint becomes unresponsive/unusable. This section investigates how disabling a joint of the robot arm impacts the performance of movement control networks.

The movement control networks were originally evolved with full control over all six joints. The networks were then evaluated on the test set to see how their movement control is impacted when a single joint was disabled (i.e. the network will try to change the joints positions as usual but one joint cannot be moved). Each network was evaluated first with only joint 1 disabled, then with only joint

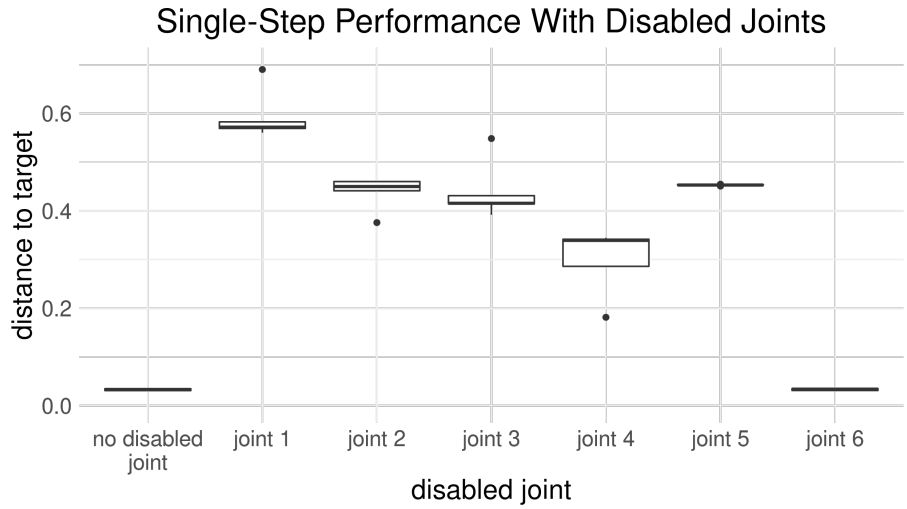

Figure 5.9: Boxplot of distance to target achieved by single-step network with different joints disabled.

2 disabled and so on.

Figure 5.9 shows the performance spread of five single-step networks and figure 5.10 shows the performance spread of five multi-step networks. The single-step network and the multi-step network are affected by disabled joints in different ways. The single-step network does not suffer much when joint 6 is disabled. This is because joint 6 only rotates the end effector and while this changes the orientation of the end effector, it does not change its position. Despite this, the performance of the multi-step network does suffer when joint 6 is disabled. Since the multi-step network receives the joint positions at each time-step, the network may be receiving state information which it is unfamiliar with. If the multi-step network usually moves joint 6 along with the other joints, disabling joint 6 may result in the network receiving a joint configuration that it doesn't know how to process/understand.

In the case of both the single-step and the multi-step networks, disabling a single joint usually has a considerable negative impact on the networks ability to move to the target. In part, this is because when certain joints are disabled

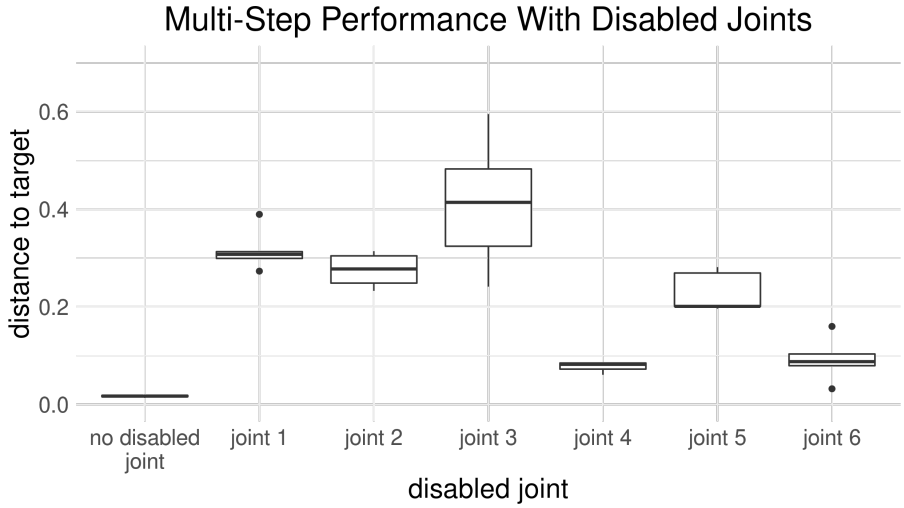

Figure 5.10: Boxplot of distance to target achieved by multi-step network with different joints disabled.

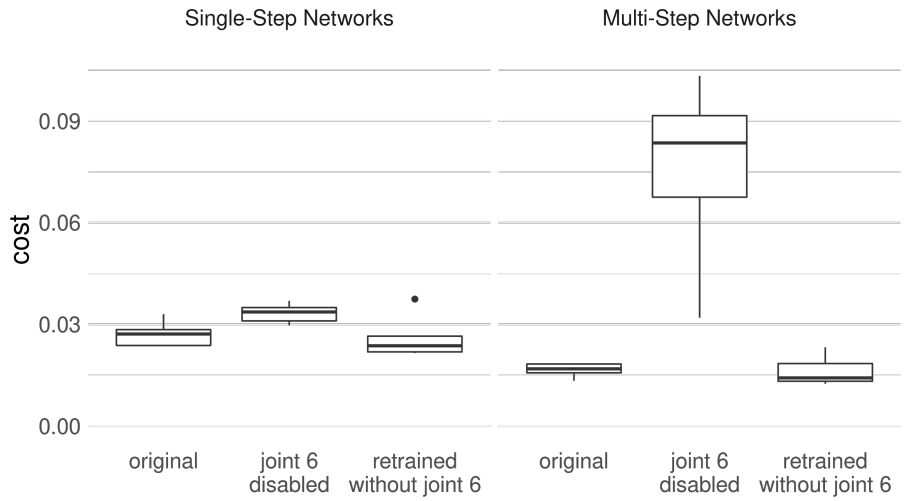

Figure 5.11: Boxplot of cost (distance to target) achieved by networks I: trained with control of all joints II: trained with control of all joints but but with joint 6 disabled at test time III: trained with control of all joints except joint 6. Results for single-step network are on the left and results for multi-step network on the right.

(namely joints 1, 2, 3 and 5), the reachable workspace of the arm is significantly reduced. For example, if joint 1 is disabled, the arm can no longer rotate about the base and a deterioration of movement performance is inevitable. On the other hand, when joint 4 is disabled, movement performance is not impacted as much (joint 4 is not used as much as the others). However, even when losing control of a joint of lesser importance, the movement control networks struggle to reach the target.

Figure 5.11 visualizes the distance to target achieved by movement control networks trained with and without control of joint 6. As expected, performance degrades when a network which was trained with control of all six joints loses control of joint 6. When the networks are retrained without control of joint 6, not only does performance return to the original level, it actually outperforms the originally trained networks. This is likely due to a reduced number of weights that need to be optimized and the fact that the networks no longer need to control an extra joint. Despite joint 6 being of lesser importance compared to the other joints, these results showcase the adaptability of CMA-ES to movement control of a robot arm.

5.5 Final Movement Network Results

Due to the computational restrictions, CMA-ES was not allowed to converge fully for many of the experiments. To give a more realistic view of the performance of CMA-ES, networks were evolved for the single-step task (with imitation seed network) and the multi-step task over 500,000 evaluations each.

On average (over the test dataset of 1000 unseen target locations), the evolved single-step controller was able to place the end effector 1.9 cm away from the target. The evolved multi-step controller was able to place the end effector 1.5

cm away from the target. 1.5 cm is close enough to the target that we can conclude that the multi-step network has successfully solved the movement control task.

5.6 Supervisor Network

This section discusses the results of evolving a supervisor network to select when to perform each of the four parts of the pick-and-place task. The supervisor

The supervisor network was evolved for 400,000 evaluations using the dataset of 20,000 (state, label) pairs described in section 3.3. The evolved network selects the correct sub-task to be performed with an accuracy of 99.5% (when evaluated on 4000 unseen (state, sub-task id) pairs).

The evolved supervisor network was used to perform 1000 episodes of the pick-and-place task using the multi-step network discussed in section 5.5. Out of the 1000 episodes, 927 were completed successfully, 33 failed due to the supervisor network selecting the wrong sub-task and 51 failed because the object could not be grasped properly due to the multi-step network placing the end effector inaccurately. The combination of the supervisor network and the multi-step network results in a system which can accurately perform the pick-and-place task.

Chapter 6

Conclusion and Future Work

In this thesis, we have demonstrated that neuroevolution is an effective approach for creating an autonomous agent that can accurately control a robot arm. Different evolutionary algorithms were compared and evaluated on the task. CMA-ES outperformed both DE and PSO at evolving the network weights to move the arm to an arbitrary target location. Furthermore, we evaluated and compared two distinct methods for how the network controls the arm (single-step and multi-step) both of which offer their own advantages and disadvantages. The multi-step approach offers more accuracy but is slower when compared to its single-step counterpart.

We also highlight the benefits of using expert demonstrations to improve the evolution process. While expert demonstrations do not ultimately improve the final networks performance, the results show that a small number of expert demonstrations can be used to improve the convergence speed of CMA-ES. The final network controller, evolved using CMA-ES and the multi-step approach, can move the arm to within 1.5cm of a randomly placed target.

The evolved supervisor network is able to use the movement network to complete the pick-and-place task with a high degree of accuracy. The movement

control network can position the arm such that an object can be grasped almost all of the time and can place the grasped object very close to the target location. The results show that using evolutionary algorithms to evolve the weights of a neural network with a simple architecture can result in an agent that can competently solve complex reinforcement learning problems.

The final pick-and-place controller consists of two very small neural networks (one supervisor network and one multi-step network). This represents a compact and lightweight system which could easily be run efficienty on an edge device to control a real-world robot.

The relatively long training times of movement control networks meant that further experimentation was limited. Training a multi-step network can take days to fully converge. This limited the number of different network architectures, control approaches and evolutionary algorithms that could be investigated.

The pick-and-place task was solved by splitting it up into sub-tasks and solving each of these sub-tasks individually. While the final pick-and-place agent represents an efficient solution to the problem, an end-to-end neuroevolution approach may be possible. An end-to-end approach which uses a single network to perform all sub-tasks in one go would pose a much more difficult optimization problem and would require considerable time and computational resources. An end-to-end approach could be tackled in future research. At the same time, the supervisor approach offers an elegant and light-weight solution to the problem.

Future work would involve applying the methods described in this report to similar tasks. The techniques were designed with the intent of making them applicable to other robot control problems. The movement control and the supervisor network could be extended to solve other robot arm tasks which involve movement such as opening a door.

We carried out a brief investigation into how a movement control network

performs when joints are disabled. However, due to time constraints, more extensive experiments (such as using a network trained with 6 active joints as a seed solution for evolving a network with only 5 active joints) could not be carried out. Designing and training a network which is robust to some joints being disabled would be an interesting subject for further research (e.g. randomly disabling some joints when networks are being evaluated).

Neuroevolution algorithms which can evolve the network topology alongside the weights could be used to improve the performance of the pick-and-place agent. Although using CMA-ES is shown to be an effective method of evolving the weights of a NN controller for movement control, advanced neuroevolution algorithms such as NEAT could potentially result in a superior solution. The NN architecture chosen for this work is quite shallow (only one hidden layer), it is unlikely that this architecture is optimal. NEAT and Hyper-NEAT could be used to find an architecture which is optimally suited to the movement task.

Overall, this thesis investigated and compared several different approaches and techniques for evolving neural network controllers. The final agent represents an impressive solution to robot arm control.

References

[1] P. Li, S.-h. Lee, and H.-Y. Hsu, "Review on fruit harvesting method for potential use of automatic fruit harvesting systems," *Procedia Engineering*, vol. 23, pp. 351–366, 2011. 1

[2] C. Liang, K. Chee, Y. Zou, H. Zhu, A. Causo, S. Vidas, T. Teng, I. Chen, K. Low, and C. Cheah, "Automated robot picking system for e-commerce fulfillment warehouse application," in *The 14th IFToMM World Congress*, 2015. 1

[3] K. Benali, J.-F. Brethé, F. Guérin, and M. Gorka, "Dual arm robot manipulator for grasping boxes of different dimensions in a logistics warehouse," in *2018 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 2018, pp. 147–152. 1

[4] B. Siciliano, O. Khatib, and T. Kröger, *Springer handbook of robotics*. Springer, 2008, vol. 200. 1

[5] M. Arntz, T. Gregory, and U. Zierahn, "The risk of automation for jobs in oecd countries: A comparative analysis," 2016. 2

[6] L. Ivančić, D. Suša Vugec, and V. Bosilj Vukšić, "Robotic process automation: systematic literature review," in *International Conference on Business Process Management*. Springer, 2019, pp. 280–295. 2

[7] R. Liu, F. Nageotte, P. Zanne, M. de Mathelin, and B. Dresp-Langley, "Deep reinforcement learning for the control of robotic manipulation: a focussed mini-review," *Robotics*, vol. 10, no. 1, p. 22, 2021. 2, 18

[8] S. Amarjyoti, "Deep reinforcement learning for robotic manipulation-the state of the art," *arXiv preprint arXiv:1701.08878*, 2017. 2, 4, 18, 19

[9] S. Gu, E. Holly, T. Lillicrap, and S. Levine, "Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates," in *2017 IEEE international conference on robotics and automation (ICRA)*. IEEE, 2017, pp. 3389–3396. 2, 3, 6, 18, 19

[10] K. O. Stanley, J. Clune, J. Lehman, and R. Miikkulainen, "Designing neural networks through neuroevolution," *Nature Machine Intelligence*, vol. 1, no. 1, pp. 24–35, 2019. 2

[11] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning," *arXiv preprint arXiv:1712.06567*, 2017. 2, 12, 13

[12] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution strategies as a scalable alternative to reinforcement learning," *arXiv preprint arXiv:1703.03864*, 2017. 3, 12, 13

[13] R. Wen, Z. Guo, T. Zhao, X. Ma, Q. Wang, and Z. Wu, "Neuroevolution of augmenting topologies based musculor-skeletal arm neurocontroller," in *2017 IEEE international instrumentation and measurement technology conference (I2MTC)*. IEEE, 2017, pp. 1–6. 3, 16, 19

[14] P.-C. Huang, J. Lehman, A. K. Mok, R. Miikkulainen, and L. Sentis, "Grasping novel objects with a dexterous robotic hand through neuroevolution," in

2014 IEEE Symposium on Computational Intelligence in Control and Automation (CICA). IEEE, 2014, pp. 1–8. 3, 16, 19

[15] D. E. Moriarty and R. Miikkulainen, "Evolving obstacle avoidance behavior in a robot arm," in *From animals to animats 4: Proceedings of the fourth international conference on simulation of adaptive behavior*, vol. 4. MIT Press, 1996, p. 468. 3, 19, 20

[16] T. D'Silva and R. Miikkulainen, "Learning dynamic obstacle avoidance for a robot arm using neuroevolution," *Neural processing letters*, vol. 30, no. 1, pp. 59–69, 2009. 3, 16, 19, 20, 29

[17] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013. 4

[18] F. Ebert, C. Finn, S. Dasari, A. Xie, A. Lee, and S. Levine, "Visual foresight: Model-based deep reinforcement learning for vision-based robotic control," *arXiv preprint arXiv:1812.00568*, 2018. 4

[19] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017. 6

[20] A. Puigdomènech Badia, B. Piot, S. Kapturowski, P. Sprechmann, A. Vitvitskyi, D. Guo, and C. Blundell, "Agent57: Outperforming the atari human benchmark," *arXiv e-prints*, pp. arXiv–2003, 2020. 6

[21] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015. 6

[22] X. Yu and M. Gen, *Introduction to evolutionary algorithms*. Springer Science & Business Media, 2010. 6

[23] K. De Jong, "Evolutionary computation: a unified approach," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 373–388. 7

[24] R. Storn and K. Price, "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces," *Journal of global optimization*, vol. 11, no. 4, pp. 341–359, 1997. 8

[25] S. Das and P. N. Suganthan, "Differential evolution: A survey of the state-of-the-art," *IEEE transactions on evolutionary computation*, vol. 15, no. 1, pp. 4–31, 2010. 9, 12

[26] J. Chakraborty, A. Konar, L. C. Jain, and U. K. Chakraborty, "Cooperative multi-robot path planning using differential evolution," *Journal of Intelligent & Fuzzy Systems*, vol. 20, no. 1-2, pp. 13–27, 2009. 9

[27] L. Lakshminarasimman and S. Subramanian, "Applications of differential evolution in power system optimization," in *Advances in Differential Evolution*. Springer, 2008, pp. 257–273. 9

[28] N. Hansen and A. Ostermeier, "Completely derandomized self-adaptation in evolution strategies," *Evolutionary computation*, vol. 9, no. 2, pp. 159–195, 2001. 9, 37

[29] [Online]. Available: https://www.ini.rub.de/PEOPLE/glasmtbl/projects/bbcomp/#results 9

[30] Y. Akimoto, A. Auger, and N. Hansen, "Cma-es and advanced adaptation

mechanisms," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, 2016, pp. 533–562. 9

[31] N. Hansen, "Injecting external solutions into cma-es," *arXiv preprint arXiv:1110.4181*, 2011. 10

[32] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4. IEEE, 1995, pp. 1942–1948. 10, 14

[33] Y. Zhang, S. Wang, and G. Ji, "A comprehensive survey on particle swarm optimization algorithm and its applications," *Mathematical problems in engineering*, vol. 2015, 2015. 12

[34] N. Jain, U. Nangia, and J. Jain, "A review of particle swarm optimization," *Journal of The Institution of Engineers (India): Series B*, vol. 99, no. 4, pp. 407–411, 2018. 12

[35] G. A. Jastrebski and D. V. Arnold, "Improving evolution strategies through active covariance matrix adaptation," in *2006 IEEE international conference on evolutionary computation*. IEEE, 2006, pp. 2814–2821. 12

[36] C. Igel, T. Suttorp, and N. Hansen, "A computational efficient covariance matrix update and a (1+ 1)-cma for evolution strategies," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, 2006, pp. 453–460. 12

[37] C. Igel, N. Hansen, and S. Roth, "Covariance matrix adaptation for multi-objective optimization," *Evolutionary computation*, vol. 15, no. 1, pp. 1–28, 2007. 12

[38] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015. 12

[39] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*. PMLR, 2016, pp. 1928–1937. 12

[40] K. Mason and S. Grijalva, "Building hvac control via neural networks and natural evolution strategies," in *2021 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2021, pp. 2483–2490. 13

[41] D. Floreano, P. Dürr, and C. Mattiussi, "Neuroevolution: from architectures to learning," *Evolutionary intelligence*, vol. 1, no. 1, pp. 47–62, 2008. 13

[42] J. P. Donate, X. Li, G. G. Sánchez, and A. S. de Miguel, "Time series forecasting by evolving artificial neural networks with genetic algorithms, differential evolution and estimation of distribution algorithm," *Neural Computing and Applications*, vol. 22, no. 1, pp. 11–20, 2013. 13, 14

[43] L. Wang, Y. Zeng, and T. Chen, "Back propagation neural network with adaptive differential evolution algorithm for time series forecasting," *Expert Systems with Applications*, vol. 42, no. 2, pp. 855–863, 2015. 13, 14

[44] A. Slowik and M. Bialko, "Training of artificial neural networks using differential evolution algorithm," in *2008 conference on human system interactions*. IEEE, 2008, pp. 60–65. 13

[45] M. Baioletti, G. Di Bari, V. Poggioni, and M. Tracolli, "Differential evolution for learning large neural networks," 2018. 14

[46] J. Ilonen, J.-K. Kamarainen, and J. Lampinen, "Differential evolution training algorithm for feed-forward neural networks," *Neural Processing Letters*, vol. 17, no. 1, pp. 93–105, 2003. 14

[47] K. Mason, M. Duggan, E. Barrett, J. Duggan, and E. Howley, "Predicting host cpu utilization in the cloud using evolutionary neural networks," *Future Generation Computer Systems*, vol. 86, pp. 162–173, 2018. 14

[48] K. Mason, J. Duggan, and E. Howley, "A multi-objective neural network trained with differential evolution for dynamic economic emission dispatch," *International Journal of Electrical Power & Energy Systems*, vol. 100, pp. 201–221, 2018. 14

[49] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE signal processing magazine*, vol. 29, no. 6, pp. 141–142, 2012. 14

[50] R. A. Fisher, "The use of multiple measurements in taxonomic problems," *Annals of eugenics*, vol. 7, no. 2, pp. 179–188, 1936. 14

[51] E. A. Grimaldi, F. Grimaccia, M. Mussetta, and R. Zich, "Pso as an effective learning algorithm for neural network applications," in *Proceedings. ICCEA 2004. 2004 3rd International Conference on Computational Electromagnetics and Its Applications, 2004.* IEEE, 2004, pp. 557–560. 14

[52] V. G. Gudise and G. K. Venayagamoorthy, "Comparison of particle swarm optimization and backpropagation as training algorithms for neural networks," in *Proceedings of the 2003 IEEE Swarm Intelligence Symposium. SIS'03 (Cat. No. 03EX706).* IEEE, 2003, pp. 110–117. 14

[53] C. Zhang, H. Shao, and Y. Li, "Particle swarm optimisation for evolving artificial neural network," in *Smc 2000 conference proceedings. 2000 ieee in-*

ternational conference on systems, man and cybernetics.'cybernetics evolving to systems, humans, organizations, and their complex interactions'(cat. no. 0, vol. 4. IEEE, 2000, pp. 2487–2490. 14

[54] A. Espinal, M. Sotelo-Figueroa, J. A. Soria-Alcaraz, M. Ornelas, H. Puga, M. Carpio, R. Baltazar, and J. Rico, "Comparison of pso and de for training neural networks," in *2011 10th Mexican International Conference on Artificial Intelligence*. IEEE, 2011, pp. 83–87. 14

[55] C. Igel, "Neuroevolution for reinforcement learning using evolution strategies," in *The 2003 Congress on Evolutionary Computation, 2003. CEC'03.*, vol. 4. IEEE, 2003, pp. 2588–2595. 14

[56] K. Mason, J. Duggan, and E. Howley, "Forecasting energy demand, wind generation and carbon dioxide emissions in ireland using evolutionary neural networks," *Energy*, vol. 155, pp. 705–720, 2018. 14

[57] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary computation*, vol. 10, no. 2, pp. 99–127, 2002. 15, 16

[58] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial life*, vol. 15, no. 2, pp. 185–212, 2009. 16

[59] K. O. Stanley, "Compositional pattern producing networks: A novel abstraction of development," *Genetic programming and evolvable machines*, vol. 8, no. 2, pp. 131–162, 2007. 16

[60] J. Clune, B. E. Beckmann, C. Ofria, and R. T. Pennock, "Evolving coordinated quadruped gaits with the hyperneat generative encoding," in *2009*

iEEE congress on evolutionary computation. IEEE, 2009, pp. 2764–2771. 17

[61] B. G. Woolley and K. O. Stanley, "Evolving a single scalable controller for an octopus arm with a variable number of segments," in *International Conference on Parallel Problem Solving from Nature.* Springer, 2010, pp. 270–279. 17

[62] E. Haasdijk, A. A. Rusu, and A. Eiben, "Hyperneat for locomotion control in modular robots," in *International Conference on Evolvable Systems.* Springer, 2010, pp. 169–180. 17

[63] J. Drchal, J. Koutník, and M. Snorek, "Hyperneat controlled robots learn how to drive on roads in simulated environment," in *2009 iEEE congress on evolutionary computation.* IEEE, 2009, pp. 1087–1092. 17

[64] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation learning: A survey of learning methods," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, pp. 1–35, 2017. 17

[65] H. Ravichandar, A. Polydoros, S. Chernova, and A. Billard, "Recent advances in robot learning from demonstration," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 3, no. ARTICLE, pp. 297–330, 2020. 17

[66] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robotics and autonomous systems*, vol. 57, no. 5, pp. 469–483, 2009. 17

[67] I. V. Karpov, V. K. Valsalam, and R. Miikkulainen, "Human-assisted neuroevolution through shaping, advice and examples," in *Proceedings of the*

13th annual conference on Genetic and evolutionary computation, 2011, pp. 371–378. 17

[68] M. W. Spong, S. Hutchinson, M. Vidyasagar *et al.*, *Robot modeling and control*. Wiley New York, 2006, vol. 3. 18

[69] G. S. Hornby, S. Takamura, J. Yokono, O. Hanagata, T. Yamamoto, and M. Fujita, "Evolving robust gaits with aibo," in *Proceedings 2000 iCRA. millennium conference. iEEE international conference on robotics and automation. symposia proceedings (cat. no. 00CH37065)*, vol. 3. IEEE, 2000, pp. 3040–3045. 19

[70] H. Lipson and J. B. Pollack, "Automatic design and manufacture of robotic lifeforms," *Nature*, vol. 406, no. 6799, pp. 974–978, 2000. 19

[71] D. E. Moriarty and R. Miikkulainen, "Hierarchical evolution of neural networks," in *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No. 98TH8360)*. IEEE, 1998, pp. 428–433. 20

[72] E. Rohmer, S. P. Singh, and M. Freese, "V-rep: A versatile and scalable robot simulation framework," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2013, pp. 1321–1326. 22

[73] S. James, M. Freese, and A. J. Davison, "Pyrep: Bringing v-rep to deep robot learning," *arXiv preprint arXiv:1906.11176*, 2019. 24

[74] *Kinematics*. London: Springer London, 2009, pp. 39–103. [Online]. Available: https://doi.org/10.1007/978-1-84628-642-1_2 26