Neuroevolutionary Planning for Robotic Control

Reza Mahjourian
Department of Computer Science
The University of Texas at Austin
Austin, TX 78712
reza@cs.utexas.edu

Doctoral Dissertation Proposal

Supervising Professor: Risto Miikkulainen

Abstract

Traditional industrial robotics is primarily focused on optimal path control. In contrast, modern robotic tasks increasingly require a greater degree of adaptability and flexibility to deal with variations in the robots environment. Moreover, new robotic tasks can be specified without precise models or target behaviors. Neuroevolution of controllers is a promising approach to such tasks: Neural networks represent nonlinear control naturally and evolutionary computation creates robust and flexible solutions. This proposal discusses completed work on evolving controllers for an object insertion task. Experiments show that, with proper input and output setup, evolution is able to discover controllers with precise behavior. However, when extending the task to require strategy and planning, finding solutions becomes hard. This proposal discusses a new evolutionary method to both discover and complete subtasks that leads to the completion of the main task. The proposed method evolves a set of action sequence networks and a planning network using an adaptation of the Enforced Subpopulations (ESP) [13] method. As a stretch goal, the proposal discusses extending this setup toward sensory-driven task specifications with the goal of eliminating the need for hand-designed objective functions.

Contents

1 Introduction

Traditional industrial robotics relies on precise task specifications and accurate body models. Legacy approaches to such robotic tasks are primarily concerned with optimal path control. However, new robotics applications require controllers that can function with imperfect sensors and actuators and operate in changing environments. Moreover, new robotics systems need to be built with adaptability and flexibility as core requirements.

Neuroevolution [40] is a promising approach to developing robotic controllers. On one hand, neural networks are well-suited for implementing nonlinear control. They are robust against noise, can function with incomplete inputs, and have been tested in many benchmarks. On the other hand, evolutionary computation is able to automatically discover topologies and connection weights for neural networks [39]. Neuroevolution is especially useful when optimal behavior is not known beforehand. In such situations, including many control tasks, training targets are not available to train a neural network using supervised learning methods.

Many robotic tasks studied in the literature are designed to have relatively large solution spaces. For example, a walk controller for a multi-legged robot [44] is evaluated based on the locomotion speed that it can achieve, but is free to use any type of gait. As another example, a controller for a robotic arm that is tasked to avoid obstacles placed in its environment [6] is free to choose any path for moving the arm. In contrast, many industrial robotic tasks require a high degree of precision, which reduces the solution space for the controller. For example, Figure 1 shows an industrial robot that is designed to insert an electrical component onto a circuit board. The controller on this robot will fail unless it is able to align the wires from the component with the holes on the board perfectly.

Additionally, most robotic tasks require completing multi-step goals, which require a planning mechanism. Traditional robotic controllers are typically hand-designed and the division of tasks into subtasks is done by a human designer. Similarly, most learning and evolutionary search methods are designed to optimize short-term reactive behaviors that are then utilized in some higher-level controller. As the tasks and operating environments for the robots are getting more complicated, there is a growing need for learning and search methods that can discover plans to complete their goals without relying on a pre-existing human-designed subtask structure.

This proposal is focused on developing new techniques and controller designs that can address the needs of the upcoming generation of industrial robotics. In order for the new controllers to replace preprogrammed factory robots they need to exhibit a comparable level of accuracy in their behavior. On the other hand, in order to have a low engineering cost, their internal organization and their training process needs to be more automated.

The remainder of this proposal is organized as follows. Section 2 reviews neuroevolution and some existing work on evolving robotic controllers. Section 3 looks at a series of experiments with gradually-increasing degrees of difficulty building toward an object insertion task. After discussing successful and unsuccessful results from completed work, Section 4

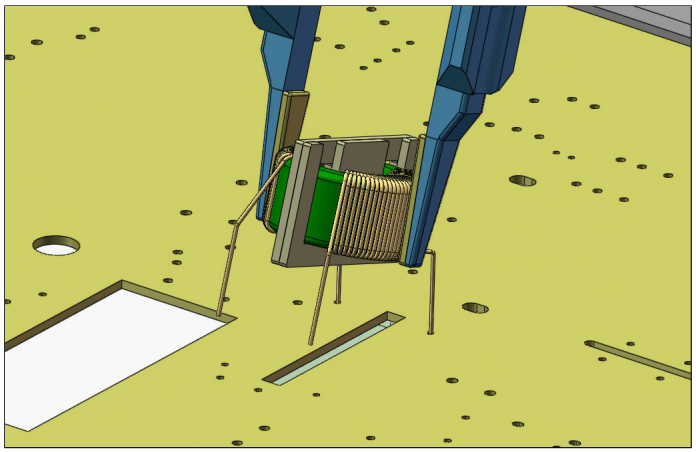

Figure 1: **Example industrial robotic task**. The robot's task is to insert an electrical component into a circuit board. Successful insertion of the component requires aligning the four wires with the corresponding holes on the board. It requires flexibility to deal with the deformations to the wires, robustness to deal with imperfect sensors and actuators, and planning for breaking the task into subtasks and completing them one by one.

proposes a new evolutionary method with a focus on planning. Also, a series of experiments are proposed to evaluate whether the proposed evolutionary method can overcome the limitations encountered in completed work. Section 5 concludes the proposal.

2 Background and Related Work

Neuroevolution [33, 37] is a method for modifying neural network weights and topologies such that the resulting neural network can function on a specific task. Neuroevolution has been applied to many robotic tasks [9] and has been used in combination with many neural network architectures [8]. The following sections give an overview of related work in this domain.

2.1 Neuroevolution

Neuroevolution [32] is an unsupervised learning method for training neural networks using genetic algorithms. Most neural-network learning methods are based on supervised learning, where there is a set of input-output examples available to describe the desired behavior. However, many real-world tasks are not amenable to supervised learning. For example, in board games and robotics, it is not always known what the optimal action at each point in time is.

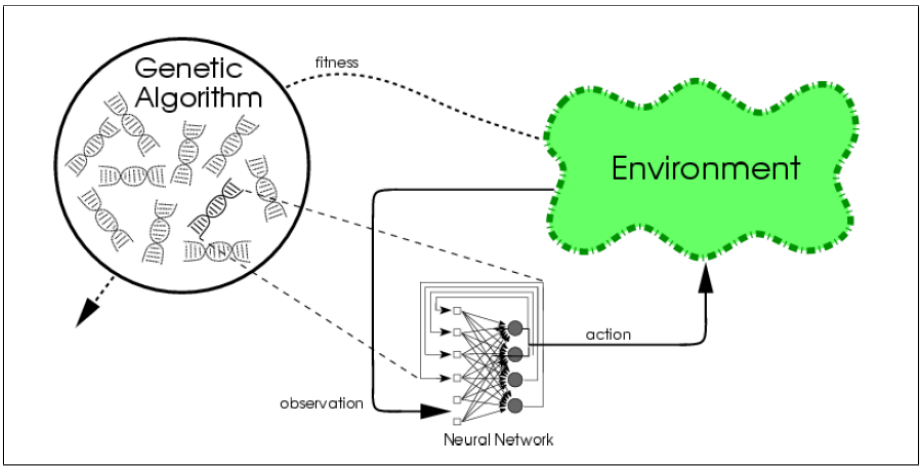

Figure 2: **Neuroevolution**, image from [32]. During each generation of neuroevolution, a genetic algorithm creates a new population of genotypes. The genotypes are translated into phenotypes, which are neural networks. The neural networks are evaluated in the task's environment. THe observations from the environment are fed to the neural network as inputs and the outputs from the neural network are passed to the environment as actions. Based on their performance on the task, the neural networks receive fitness scores. The genetic algorithm uses the fitness scores from the current generation to select candidate genotypes for creating the population for the next generation using crossover and mutation.

Figure 2 shows an overview of neuroevolution. The process starts by creating a population of neural networks encodings. The encodings, which are referred to as *genotypes*, are method-specific. They can be as simple as a set of real numbers encoding the connection weights or as complex as a set or rules whose application will create the neural network [3]. During each iteration of neuroevolution, every the genotypes in the population is translated into a *phenotype*, which is a neural network with specific connections and weights. The phenotypes are then evaluated on the given task. They are tried out in the environment for a set amount of time, a set number of time steps, or until some condition is met in the environment. Based on how well they do, each phenotype receives a fitness score, which is assigned back to its original genotype. Once all genotypes have fitness scores, the best performing genotypes are picked according to some selection algorithm. Then, a genetic algorithm applies mutation and crossover operations among the selected genotypes to create the next generation of the population. This process continues until a genotype with a desired level of fitness is discovered. This genotype is often referred to as the *champion*. Alternatively, the process can terminate when the maximum number of iterations is met, signaling a partial success or failure depending on the task.

2.2 Neuroevolutionary Robotic Tasks

This section reviews a few examples of the robotics domains where neuroevolution has been employed.

2.2.1 Pole Balancing

Pole balancing (inverted pendulum) is an established benchmark robotic task. The task's environment consist of a pole hinged on top of cart which can move back and forth on a limited track. In the simple case, the objective is to apply a force to the cart at regular intervals such that the pole stays balanced and the cart does not exit the track boundaries. In the simple case, the controller receives the position of the cart, velocity of the cart, angle of the pole, and angular velocity of the pole. There are also modified versions of the task that make it more complicated, either by limiting the amount of state information that is observable by the controller, or by hinging additional poles on top of the first pole.

Gomez et al. [15] compared the performance of neuroevolution with other learning algorithm such as Policy Gradient RL, Q-learning, and Sarsa(λ) and found neuroevolution to perform considerably better on this task. Pole balancing is an interesting problem since it is an unstable control system and serves as a good surrogate for more complex robotic tasks. On the other hand, it is a reactive task, meaning that solving the problem requires responding to the current situation only and does not involve planning for future goals.

2.2.2 Locomotion

Among the earliest works on evolutionary robotic controllers was the work by Beer et al. [2] who introduced a neural architecture for locomotion of a hexapod simulated robot. This architecture was designed based on studies of cockroaches. The legs had fixed knee joints and each leg was controlled by two neurons responsible for the leg swing and elevation. Each leg also had a pacemaker neuron. The pacemaker neurons from the six legs were cross-connected to allow for coordination between the legs. They used a two-stage evolutionary process where first the behavior of a single leg was evolved, and then the coordination of the legs was evolved to create a walk. This architecture was also tested on a real robot [29].

These results show that neuroevolution can solve the locomotion problem, which is a complex unstable control problem. On the other hand, locomotion has a larger solution space compared to an object insertion task. The controller can choose from a variety of gaits. Also, a suboptimal gait may result in a slower walk, rather than a complete failure.

2.2.3 Navigation

Another early application of neuroevolution was in robot navigation [10]. Floreano et al. experimented with a physical wheeled robot in a small maze with the objective of maximizing the robot's speed without colliding with the maze walls. The robot had two

wheels and eight infrared sensors to measure the distance to nearby obstacles. Using a simple genetic algorithm over 200 generations they evolved the weights of a neural network with a hand-designed topology. The final network was able to successfully navigate the robot around the maze at about 25 percent of its maximum speed. This work was one of the earliest examples of running the entire evolution process on a physical robot.

2.2.4 Finless Rocket Flight

Gomez et al. [16] applied neuroevolution to stabilize flight of a finless version of the Interorbital Systems RSX-2 sounding rocket. Without the extra drag from the fins, the rocket was able to fly much higher for the same amount of fuel. However, without fins the rocket's center of pressure fell in front of its center of gravity and created an unstable system. The neural network received the rocket's orientation (pitch, yaw, roll), the rate of change for each orientation component, and the rocket's altitude and velocity. The outputs from the neural network controlled the throttle on each of the four engines.

The rocket flight problem is another example of a reactive unstable control task with similarities to the pole balancing problem.

2.2.5 Vehicle Collision Warning

Kohl et al. [25] developed a vehicle collision warning system on a simulated race track. As input, the network was given readings from seven simulated rangefinders covering non-overlapping sectors in front of the car. Each rangefinder would indicate the distance to the closest obstacle or curb in its viewing sector to the neural network. The network was expected to produce a real-valued output predicting the estimated time to the next collision. This prediction was to be used a warning signal to prevent crashes. Successful networks were evolved that could accurately predict collision times. They also experimented with using a 20×14 grayscale image as the input to the network, which was successful as well.

Their success with using raw visual input showed that it is possible to use neuroevolution on problems with high-dimensional inputs. However, using the simulated rangefinders performed better. These results showed that neuroevolution can benefit from higher-level inputs.

2.2.6 Controlling a Robotic Arm

Neuroevolution has been used to control an OSCAR-6 robotic arm [34, 7] in a simulator. The evolved neural network could take the end-effector to a requested position by controlling the position of the joints. Also, they could evolve a controller that avoided obstacles placed between the end-effector and its target. The robotic arm had three links and six degrees of freedom. The inputs to the neural network consisted of the (x, y, z) distance to the target and readings from six directional proximity sensors placed on the end-effector. The network outputs determined the positions (angles) of the joints.

The average error distance for the evolved controller was about 4cm from the target, which shows that it is difficult to evolve the precise behavior required to solve this task.

The obstacle avoidance requirement makes this task a strategic problem which needs some planning. To handle obstacle avoidance, they decomposed the task into two subtasks and evolved separate controllers for each subtask. The first subtask consisted of moving the end-effector relatively close to the target position while avoiding the obstacles. The second subtask concerned with reaching the target more precisely. The need for using two controllers for this task demonstrates that it is easier for neuroevolution to evolve reactive behaviors than long-term strategic behaviors.

2.2.7 Predator and Prey

In the predator-prey (pursuit-and-evasion) domain, a number of predators and a prey live on a discrete grid-like world. The predators try to capture the prey and the prey tries to escape the predators. During each time step all agents pick an adjacent square to which they want to move and all the moves take place at once. This is an interesting domain because a successful strategy for the predators requires cooperation of multiple agents. Also, in this domain it is possible to coevolve the predators and the prey simultaneously. This setup often leads to an evolutionary arms race where the predators and the prey try to exploit and overcome the latest strategy by the other side. Among the latest works on applying neuroevolution to this domain was the work by Jain et al. [24]. They showed that in this domain cooperation and communication evolve naturally since they were advantageous to the behavior of the predators.

The object insertion task discussed in this proposal is a single-agent problem and the multi-agent ideas do not apply to it directly. However, as discussed in Section 4, it is possible to create multi-component controllers where the components collaborate for completing the task. Doing so allows the components to take up specialized roles.

2.3 Neural Network Models

Artificial neural networks are modeled after biological neural networks. The most commonly-used neural network models are based on the perceptron model [36]. Other notable models include the spiking neuron models [30, 23], and GasNets [21]. While in the perceptron model each neuron outputs a single value representing the neuron's firing rate, in spiking neuron models each neuron generates a train of spikes whose number and timing carry the information produced by the neuron. GasNets are standard neural network models augmented with another signaling system based on diffusion of a virtual gas. In addition to the standard activations, GasNet neurons can also trigger diffusion of a virtual gas which can modulate the behavior of other nearby neurons.

This section discusses the properties of some neural network models commonly used for neuroevolutionary robotic tasks.

2.3.1 Feedforward Networks

In the simplest case, the neural network can consist only of feedforward connections. Feedforward networks are able to model Markov Decision Processes (MDPs). A process has the Markov property if its future states depend only on the present state (observations), and not on the sequence of states that precede the present state. A controller using a feedforward neural network can exhibit reactive behaviors. This is because at each activation the neural network's output is determined entirely by the inputs it receives from the present state. So, the controller is always reacting to the present situation.

Feedforward networks have the advantage that their topology is simpler than other alternatives. When used with neuroevolutionary methods that modify the network's topology, lack of recurrent connections limits the evolution's search space, thereby speeding it up. Also, if training is needed after the evolution process, standard backpropagation techniques can be used to train feedforward networks.

2.3.2 Recurrent Networks

Recurrent neural networks, on the other hand, have the ability to maintain information inside the neural network. They are able to model Partially Observable Markov Decision Processes (POMDPs). In POMDPs, the agent's present state (observations) offer a partial view of the agent's underlying state. For such tasks, the memory behavior offered by recurrent neural networks provides an advantage. Controllers using recurrent neural networks can exhibit more complex behaviors resembling planning and cognition.

2.3.3 Continuous-Time Recurrent Neural Networks

Continuous-Time Recurrent Neural Networks (CTRNNs) [1] are an example of neural networks with dynamic neuron models. Whereas in regular artificial neural networks each neuron behaves as a static function mapping its inputs to an output, in CTRNNs each neuron's output is determined by a differential equation. More specifically, in a CTRNN consisting of n dynamic neurons, the activation y_i of neuron i is determined by the differential equation

$$\tau_i \dot{y}_i = -y_i + \sigma(\sum_{j=1}^{n} w_{ij} y_j - \Theta_j) + I_i, \tag{1}$$

where τ_i is the time constant of neuron i, y_i is the activation of neuron i, $\dot{y}_i$ is the rate of change of activation of neuron i, w_{ji} is the weight of connection from neuron j to neuron i, $\sigma(x)$ is the Sigmoid activation function, y_j is the activation of neuron j, Θ_j is the bias of neuron j, and I_i is the input to neuron i.

CTRNN nodes are able to create oscillating outputs and therefore are suitable choices for modeling rhythmic behaviors like locomotion.

2.4 Network Architectures

This section reviews the types of neural network architectures that have been used for neuroevolution of robotic controllers.

2.4.1 Monolithic Architectures

In a monolithic controller, a single neural network is responsible for all sensors and all actuators on the robot. Typically, neuroevolution is applied to a fixed neural network topology with a hand-designed set of connections between neurons. The network can have one or more hidden layers in various configurations. In this case, neuroevolution's objective is to find the optimal weights for the fixed connections.

When using methods that include mutation of the topology, neuroevolution is typically allowed to insert hidden nodes in arbitrary parts of the neural network. Allowing arbitrary topologies enables neuroevolution to create fine-grained neural networks that have complex connections only in parts where they are useful. On the other hand, lack of structure can slow down the evolutionary search and make it more difficult to evolve symmetrical and synchronized behaviors.

2.4.2 Distributed Architectures

An alternative approach to monolithic networks is to use an architecture based on a set of subnetworks or modules each of which is responsible for a local group of sensors and actuators. For example, in a multi-legged robot, a subnetwork could handle the sensors and actuators for a single leg. To couple the subnetworks together and enable coordination, additional connections can be established between the internal neurons of different subnetworks.

Tellez et al. [43] used a distributed architecture to evolve a walk for the Sony Aibo robot. The Aibo has three joints per leg, resulting in 12 degrees of freedom for the walk. They evolved 12 subnetworks each consisting of a single sensor reading the current joint position (angle) and a single actuator which set joint velocities for an underlying PID controller. First, they evolved the individual joint subnetworks to exhibit oscillatory patterns similar to those observed in dogs. After evolving the individual subnetworks, the connections between the subnetworks were evolved to create a walk.

Distributed architectures reduce the complexity of the neural network by reducing the number of connections. All-to-all connections between sensors and actuators are used only locally. This reduction in complexity reduces the number of parameters that need to be optimized by neuroevolution.

2.4.3 Modular Architectures

Modular architectures exploit the symmetry in the robot's hardware or in the controller's overall design. In a modular architecture, the controller is constructed by assembling replicas of a network fragment or module. Coordination between replicas is done by having each replica receive a different set of inputs, or by parameterizing the replica's internal representation.

An example of a modular architecture is the four-legged locomotion controller by Valsalam et al. [44]. This architecture used four replicas of a module to control the four legs of the robot. The module received the positions (angles) of all four legs and output an angular velocity for its attached leg. They were able to evolve various gaits by controlling the phase relations between the module replicas. The phase relations were controlled by rearranging the inputs on individual modules so that the modules would receive different permutations of the four inputs from the four legs.

Compared to distributed architectures, modular architectures reduce the network complexity even further. Since in modular architectures the neural network is created by substituting replicas of a single module in the controller, there are even fewer weights for neuroevolution to optimize. In contrast, in distributed architectures each module has its own set of weights which need to be optimized by neuroevolution.

2.5 Methods

This section discusses different search methods used for evolving neural network controllers for robots.

2.5.1 Conventional Neuroevolution

Conventional Neuroevolution (CNE) works by evolving the connection weights in a population of neural networks [33, 37, 46, 47]. CNE is suitable for networks with a fixed topology. In this method, the genotype is a set of real values encoding the weights of all the connections in the network in some predetermined order. Each individual in the population contains values for all the connection weights. Therefore it is straightforward to create a phenotype (a full neural network) from each genotype. At each iteration, CNE evaluates all neural networks on the given task and assign them fitness scores. The next population is created using random crossover and mutation operations over the best individuals from the last generation. In CNE, crossover generates an offspring from two parents by inheriting subsets of the connections weights from one of the two parents at random. Once the offspring is generated, it is mutated by adding a small positive or negative random value to some of its connection weights. Typically, the random value is selected from a Gaussian distribution with mean of zero and a fixed standard deviation. The search continues through generations until an individual with the desired behavior is found.

2.5.2 NEAT

NeuroEvolution of Augmenting Topologies (NEAT) [40] is a method that evolves the neural network's topology as well as its connection weights. NEAT can start with a minimal topology and gradually complexity the neural network by adding new nodes and new connections over generations. Also, NEAT partitions the population into different species to maintain diversity in the population. A distance metric between the neural networks is used to determine the different species. Maintaining different species helps with preventing the population from converging to a local optimum.

NEAT is a good neuroevolutionary method, particularly in tasks that require discovering strategies for multiple goals simultaneously. Stanley et al. [41] used NEAT to evolve a foraging agent that could successfully balance multiple goals such as foraging, attacking, and retreating simultaneously.

2.5.3 CMA-ES

While the random crossover and mutation generally move the population in the direction of increasing fitness, they can be slow and inefficient. If the optimal value for some connection weight is far from the initial value, it takes many mutations and many generations for the connection weight to reach it.

Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [19] solves this problem by choosing better candidates for mutations. It analyzes the performance of parameter values tried in the previous generation to decide the direction and magnitude of mutations for each parameter in the next generation. An example of using CMA-ES for neuroevolution is the work by Igel [22] which evolved neural networks for pole balancing tasks.

CMA-ES speeds up the evolution by using more intelligent mutations. Guiding the mutations allows neuroevolution to invest its resources on trying parameter values that are more likely to improve fitness. It is straightforward to use CMA-ES with conventional neuroevolution since all networks share the same connection weights and each connection weight can be represented by a CMA-ES parameter. However, it is more tricky to use CMA-ES with methods like NEAT that modify the network topology as well. In such methods, the networks in the population may have different topologies, making it difficult to establish a mapping between connection weights of different networks.

2.5.4 SANE

Instead of maintaining a population of complete networks, Symbiotic Adaptive Neuroevolution (SANE) [35] maintains a population of network fragments consisting of hidden neurons together with their attached input and output connections. Each hidden neuron has incoming connections from all input nodes and outgoing connections to all output nodes. At evaluation time, a complete neural network is constructed by putting together a random subset of the network fragments in the population. The fitness of each network fragment is

determined as the average fitness of the neural networks in which it was used. By allowing evolution to work at a finer granularity, SANE can detect and preserve useful network fragments and speed up the search for a complete controller.

2.5.5 ESP

Enforced Subpopulations (ESP) [17] offers an improvement over SANE by partitioning the network fragments into groups. Each group corresponds to a particular subset of some predetermined network topology. Typically, the fragments correspond to hidden neurons and their attached connections. At evaluation time, a complete neural network is constructed by sampling one neuron from each subpopulation and placing them in their designated spot. Using subpopulations allows ESP to evolve specialization and cooperation between network fragments.

While SANE evolves a single population of neurons, each of which can replace any neuron in the network, ESP evolves separate subpopulations of neurons which can only be used in their designated place in the neural network. Therefore, ESP is a cooperative coevolution method. With ESP, subpopulations tend to converge to specialized roles that result in high fitness when used together with other specialized subpopulations. Such cooperation is necessary for evolving recurrent neural networks. In addition, while SANE can only be applied to neural networks where all hidden units have the same inputs and outputs, ESP can be applied to neural networks with arbitrary topologies.

While ESP was initially designed to evolve a single neural network, multi-component and multi-agent extensions of ESP have been proposed. Yong et al. [48] proposed Multi-Agent ESP and evaluated it in the predator-prey domain. They used different subpopulations for each predator agent and showed that ESP could evolve cooperative behavior between predators to capture the prey. These experiments suggest that ESP can be extended to other setups requiring coevolution of subpopulations.

As part of proposed work, Section 4 discusses an extension of ESP to evolve multiple components collaborating in a single controller to solve robotic tasks requiring planning.

2.5.6 CoSyNE

Cooperative Synapse NeuroEvolution (CoSyNE) [14] extends the idea of ESP to the level of individual connection weights. In CoSyNE, the number of subpopulations is equal to the number of connection weights in the predetermined network topology. Gomez et al. [14] experimented with using CNE, CMA-ES, NEAT, SANE, ESP, and CoSyNE on the pole balancing task and found CoSyNE to require fewer evaluations than the other methods to find a solution.

CoSyNE is particularly useful in single tasks with high accuracy requirements.

2.5.7 Multiphase Evolution

Multiphase evolution is a common technique for reducing evolution's search space. Instead of evolving all connection weights at the same time, multiphase methods optimize the weights in phases.

As an example, when evolving a hexapod locomotion Beer et al. [2] evolved the behavior of a single leg first. Freezing those weights, in the second phase they evolved the weights controlling the connections between the six legs.

Likewise, Tellez et al. [43] evolved a four-legged walk for the Aibo robot in three phases. First they evolved three subnetworks controlling the three joints on a single leg. The objective in this phase was to produce oscillatory patterns for the three joints matching the joint movements observed in dogs. In the second phase, they evolved connections between the same joints from different legs. In this phase the objective was to establish desired phase relations between different legs. In the last phase, they evolved the connections between the three joints in the same legs. The objective for the final phase was to maximize the distance traveled by the robot, thereby creating a good walk.

Multiphase evolution can speed up the evolutionary search by focusing on a small subset of all parameters in each phase. However, its application in part depends on existence of a decomposition of the original task into simpler subtasks with their corresponding objectives. Therefore it cannot be applied if such a manual task decomposition is not available or desirable. Since this proposal focuses on developing methods that can automatically discover useful subtasks, the proposed work does not use multiphase evolution.

2.5.8 Evolution and Learning

The methods discussed so far in this section discover the connection weights through random mutations and without any supervised training. However, it is possible to combine neuroevolution with learning [20].

One example of such methods is NEAT+Q, developed by Whiteson et al. [45]. In their work, the neural network served as a function approximator for Q-Learning. They used NEAT to evolve the topology and connection weights for a population of neural networks. When evaluating the networks, they trained them on some reinforcement learning task using Q-Learning. The reward received by the agent on that task was recorded as the network's fitness. Therefore, evolution was used to discover networks that were suitable for learning a task, instead of networks that already knew how to do the task.

Once the network is trained on a task using supervised learning or reinforcement learning, it is possible to overwrite the weights in the genotype with the weights resulting from training [18]. This would amount to simulating Lamarckian evolution. While this technique can improve the fitness of individual networks, it has the drawback of reducing the diversity in the population and potentially stagnating the evolutionary search.

The proposed work in Section 4 does not discuss using learning since it is not central

to the ideas developed in this proposal. However, future work can evaluate the impact of combining evolution with learning in the object insertion domain.

2.6 Treatment of Subtasks

Unlike simple benchmark tasks like pole balancing, more realistic control tasks often require some kind of coordination or planning over a set of subtasks to complete the original task. This section reviews some approaches to building controllers that can handle subtasks.

2.6.1 Disjoint Controllers

The most straightforward approach to handling complex tasks is to decompose the original task into subtasks manually and evolve disjoint specialized controllers that can handle only their corresponding subtasks.

An example of this approach can be seen in the robotic arm controller designed by Moriarty et al. [34]. The robotic arm's objective was to place an object in a given target position while avoiding obstacles. They decomposed the task into two subtasks each having its own controller. The first controller was responsible for obstacle avoidance and could bring the end-effector close to the target position. Once this step was complete, the second controller would be triggered to take the end-effector to the target position under more precise control.

Another example is the work by MacAlpine et al. [31] on evolving skills for a RoboCup simulation soccer team. They designed a set of skills for the robots like turning, walking, and kicking the ball. Each skill was optimized by evolving a sequence of trajectory points for the robot's joints. The decisions on which skill to activate were made by a hand-designed strategy module. In effect, the skill set constituted a partitioning of the original task, which was winning the soccer game, into a set of hand-designed subtasks, which were specialized skills.

When applicable, using disjoint controllers can speed up neuroevolution by reducing the complexity of behaviors required of individual controllers. In particular, if the subtasks are simple enough, it can eliminate the need for evolving planning behavior. However, using disjoint controllers depends on existence of a task decomposition. Therefore it cannot be applied if such a decomposition is not available or desirable. For example, the object insertion task in Section 3.4 requires aligning the object first and then lowering it. While this manual task decomposition makes sense, it is not the only possible decomposition and it is not necessarily the best. Therefore it is desirable to have evolutionary methods that are free to discover their own useful task decompositions. In addition to being automated, such task decompositions might also be more efficient for neuroevolution.

2.6.2 Hierarchical Skills

Lessin et al. [28] proposed a new evolution method based on hierarchical skill sets. They evolved controllers for virtual creatures which could move by contracting and extending their muscles. Their goal was to evolve a complex *fight-or-flight* behavior for the creatures. They started by evolving a few basic skills. Once these behaviors appeared, the neural networks that implemented the behaviors were frozen and encapsulated as a primitive skill. These encapsulated skills were available to and could be triggered by higher-level skills.

More specifically, in the first level they evolved three basic behaviors of moving forward, turning to the right, and turning to the left. On the next level of complexity, *turn-to* and *turn-from* behaviors were evolved. These behaviors could make the virtual creature reorient itself to face another entity or to turn away from it. These behavior were implemented by properly triggered the *turn-right* and *turn-left* primitives evolved in the first layer. Subsequent layers added more complex behaviors like *move-to*, *strike*, *attack*, *retreat*, and finally *fight-or-flight*. The hierarchy of skills leading to the fight or flight behavior was manually designed, but the implementation of the skills was done through evolution.

The skill hierarchy used in this method makes evolution of complex behavior possible and tractable. However, since the skill hierarchy is designed manually by a domain expert, it has the same limitations as discussed in the previous subsection.

2.6.3 Multiobjective Evolution

In the context of video games, Schrum et al. [38] proposed a method for evolving multiobjective behavior. They carried out experiments in the domain of Ms. Pac-Man, where the agent needed to balance two objectives of eating pills and eating ghosts to gain points and advance to the next level. They proposed an extension of NEAT where the agents received separate fitness scores for each objective. The selection algorithm took into account the performance of the networks on all objectives to determine their selection probabilities.

They also proposed an architecture for evolving multimodal behavior where the agent could switch between two or more different behavior modules at any time. The neural network had multiple sets of output nodes corresponding to different modes or subtasks. Since the output nodes for different modes were embedded in a single neural network, they could share common features captured in the network's hidden nodes.

They experimented with a human-designed task division where the current subtask and the current set of active output nodes was determined based on whether the ghosts were active or passive. When the ghosts were active and threatening, the first output module was in control. After eating a power pill and rendering the ghosts edible, the second output module would get activated and control the agent until the ghosts turned threatening again. This human-designed task division produced an agent with acceptable behavior.

They also tried allowing neuroevolution to come up with a task division by itself. They added a special mutation operation that would duplicate an existing output module into

a new one, thereby increasing the network's supported modes of operation. In addition to the outputs needed for the task, each module had an extra output neuron called a preference neuron. At each time step, the module whose preference neuron had the highest activation would be given control of the agent. The preference neurons allowed the modules to collaboratively coordinate the agent's multimodal behavior. Their results showed that a task division decided by neuroevolution was superior in performance to a human-designed task division.

There are parallels between the function of multiple objectives in this work and function of subtasks in the proposed work. However, they are orthogonal and can be used together, which is discussed as part of an experiment in Section 4.3. Also, the work discussed in this section uses a monolithic architecture where all the modules are attached to the same neural network. In contrast, the subtask networks discussed in Section 4 are separate entities and are evolved in separate subpopulations. It is expected that use of separate subpopulations can improve the evolution's performance.

3 Completed Work

In this section, a series of tasks is designed with gradually increasing degrees of difficulty to build toward a controller which can perform a real-world object insertion task. The object insertion task constitutes a good benchmark since it comes from the industry where it is a real challenge. Also, it serves as a stepping stone for more complex behaviors.

The first task discussed in this section is moving the robot's end-effector to a given location in space. The second task is inserting an object that is already grabbed by the robot's end-effector into a specified location. The third task is to grab the object first and then insert it into a specified location.

Even though neuroevolution is an automated process, there are many degrees of freedom in designing the controller and the neural networks. Among the choices that need to be made is the encoding of neural network inputs and outputs. In the simplest case, the neural network can be fed raw signals from the sensors. However, it is also possible to preprocess the raw data and prepare higher-level inputs that could be more efficient for the task. Likewise, the network outputs can be used to directly control the joints of the robot. Alternatively the network outputs can be interpreted as high-level control signals for a lower-level controller. The experiments in this section were designed to help understand the impact of such choices on efficiency of neuroevolution for learning tasks with small solution spaces.

3.1 Experiment Setup

The first experiments were done on a model of the Atlas robot using the Gazebo simulator. Due to the limitations of Gazebo for remote execution in a computer cluster, later experiments were carried out using the V-REP simulator [11].

For all object manipulation tasks, a 7-DOF (Degrees of Freedom) robotic arm with a detachable gripper as shown in Figure 3 were used. Also, the Reflexxes Motion Library [26] was used for producing online smooth trajectories. Neuroevolution was done by a python implementation of NEAT [40]. Since the simulations were time consuming, the NEAT implementation was modified to parallelize evaluation of the population in a computer cluster.

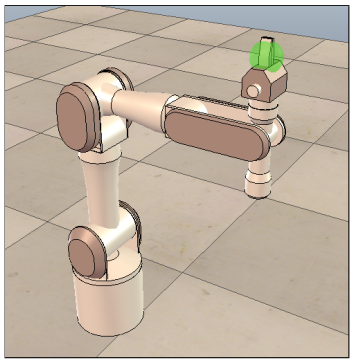

Figure 3: **7-DOF robot with detachable gripper**. This robot serves as a simplified version of the industry robot that needs to do the component insertion task. Using the simple robot with simpler meshes allows for faster simulations and more rapid experimentation.

3.2 Target-Reaching Experiment

The environment for the target-reaching task is shown in Figure 4. The controller is expected to move the cube that is attached to the suction cup on the end-effector to the target location indicated by the wireframe cube. The orientation of the cube is not important; only its distance from the target location is used in calculating the error and fitness. This task effectively requires solving the Inverse Kinematics (IK) problem. This problem is defined as calculating the correct joint angles on a multi-link object such that the end-point of the last link is positioned at a desired location.

The target-reaching experiment is a good surrogate for the industry task discussed in Section 1 because it requires precise control over all seven joints. It is also general and easily available in the simulator.

During every episode, the wireframe cube shown in Figure 4 goes through 10 random positions. The robot is given one second to catch up every time the target cube moves. Then, over the following two seconds, samples are taken from the cube's position. The error over 10 random target positions is computed as the mean of squared distances from the cube to the target location over them. Fitness is then calculated as

$$\text{Fitness} = \frac{100}{1 + \dfrac{\displaystyle\sum_{s \in \text{Samples}} \text{distance}^2(\text{Cube}_s, \text{Target}_s)}{|\text{Samples}|}}. \tag{2}$$

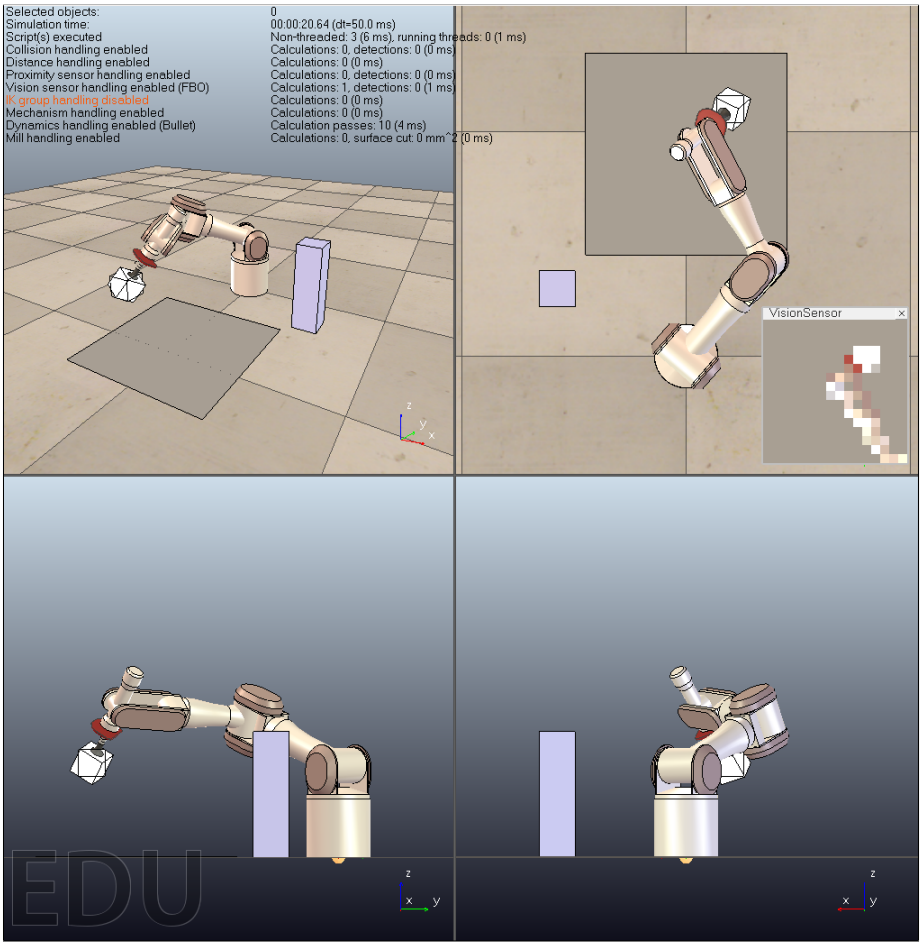

Figure 4: **Different views of the target-reaching task**. The controller needs to properly position each of the seven joints in the robot so that the white cube attached to the end-effector is aligned with the wireframe cube indicating the target position. Target-reaching serves as a surrogate for the component insertion task.

Every neural network is evaluated for 10 episodes and its fitness is calculated as the average fitness over them. The neural network's three inputs are the (x, y, z) differences

of the cube's current distance from the target location. The neural network has seven outputs, corresponding to the seven joints in the robotic arm. Figure 5 shows the overall design of the controller for the target-reaching experiment.

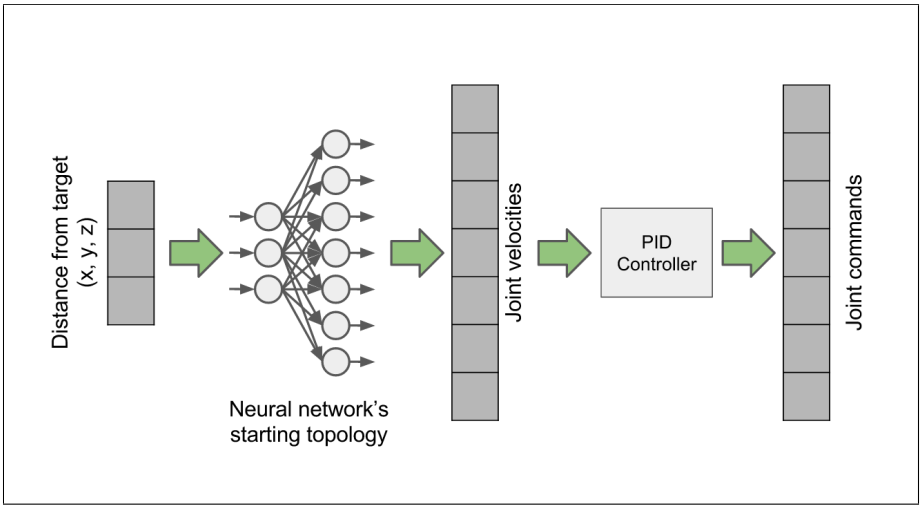

Figure 5: **Controller for the target-reaching experiment**. The neural network receives the (x, y, z) distance components to the target and outputs velocity commands for the seven joints on the robot. The requested velocities are passed to a standard PID controller to be executed.

Experiments were done with using the output of the neural network to control the robot's joints in position control, velocity control, and torque control. For position control and velocity control, a standard PID controller was used to carry out the requested position or velocity commands. As Figure 6 shows, in this task velocity control performs better than the other two control modes. For all plots in this section, the evolution process was run nine times and the average and standard error metrics were computed over the nine trials.

As discussed in Section 2, different models of neural networks have been used with neuroevolution. For the target-reaching task, experiments were done with feedforward, recurrent, and CTRNN networks. Figure 7 shows the results of evolving a target-reaching controller with different network architectures. The plot shows that feedforward and recurrent networks perform best. One reason why they perform similarly may be because the task is reactive and does not benefit from the memory in recurrent neural networks. A potential explanation for the lower performance of CTRNN networks is that the neurons in a CTRNN produce oscillating patterns that are not optimal for this non-periodic task.

Another setting that can impact the evolutionary search process is the choice of initial network topology. Starting the search with a more tightly connected topology has the

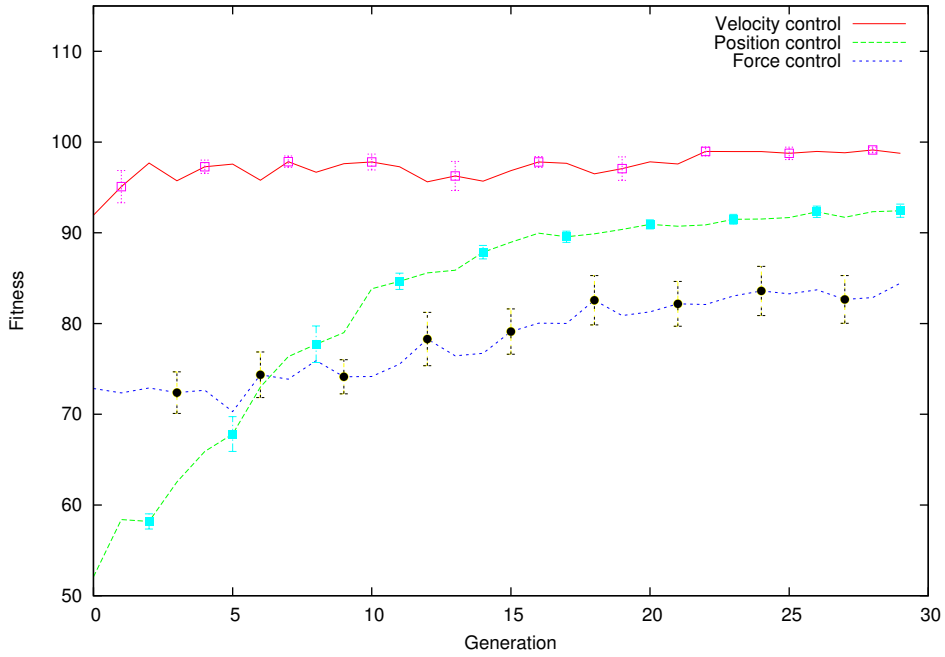

Figure 6: **Impact of control mode on performance of target-reaching controller**. All three controllers used feedforward networks with a fully-connected starting topology. Velocity control performed better compared to the other two control modes.

advantage in establishing the necessary pathways from input nodes to output nodes. On the other hand, a complex topology creates more parameters for evolution to optimize. The results in Figure 8 shows that starting with a fully-connected topology benefits the search in this task. This result is expected since in this task most joints are interacting with multiple spatial dimensions. A fully-connected initial topology already includes the necessary connections for capturing these interactions. The other compared settings are partially-connected and minimally-connected. In a partially-connected topology with n inputs and $m < n$ outputs, the ith input is connected to the jth output such that

$$j \equiv i \pmod{n}. \tag{3}$$

In a minimally-connected topology, every output is connected to a random input. If there are more inputs than outputs, a minimally-connected topology leaves some inputs unconnected at the start.

On this task, neuroevolution is able to achieve the maximum fitness of 100.0 on some trials. The simulation replays of the champion networks demonstrate that the network

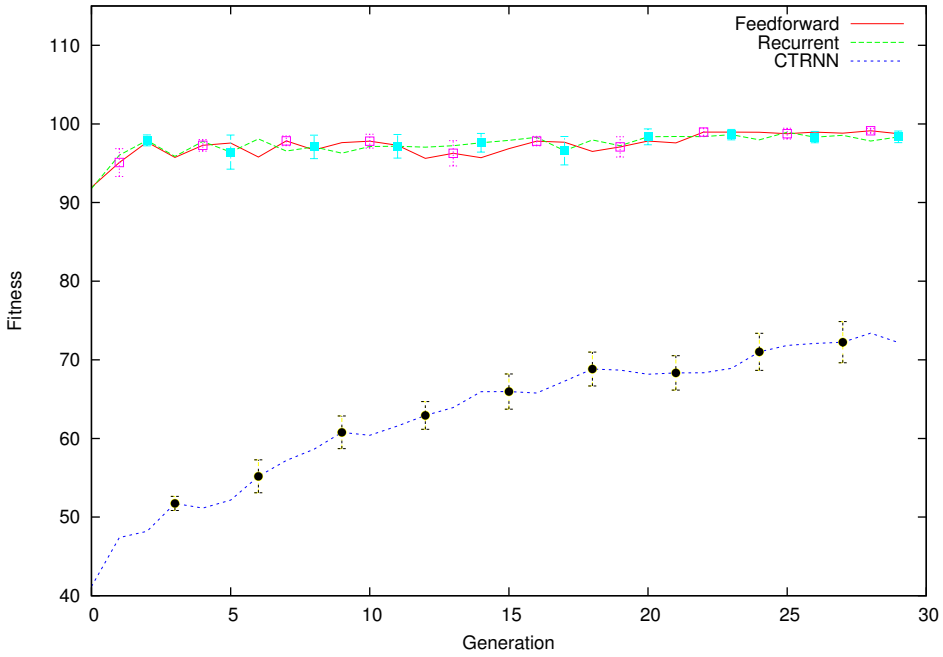

Figure 7: **Impact of neural network type on performance of target-reaching controller**. All three controllers used joint velocity control and a fully-connected starting topology. Feedforward and recurrent networks performed better than CTRNN networks. One reason why feedforward and recurrent networks performed similarly may be because the task is reactive and does not benefit from memory in recurrent networks.

is able to place the cube at the center of the target location quickly. This result is an improvement over prior work [7] where the average error distance was about 4cm from the target.

The target-reaching experiments in this section show that, under proper conditions, evolution is able to discover controllers with precise behavior. The next section discusses experiments for evaluating the robustness of evolved target-reaching controllers.

3.3 Robustness Experiments

Real-world robots have imperfect sensors and actuators. To test the evolved controller's robustness against failures experiments were done with creating an artificial fault in one of the seven joints in the robotic arm. At each time step, there was a 50% probability that the faulty joint would malfunction. During the malfunction, instead of the executing the velocity command requested by the neural network, a target velocity of zero would be set

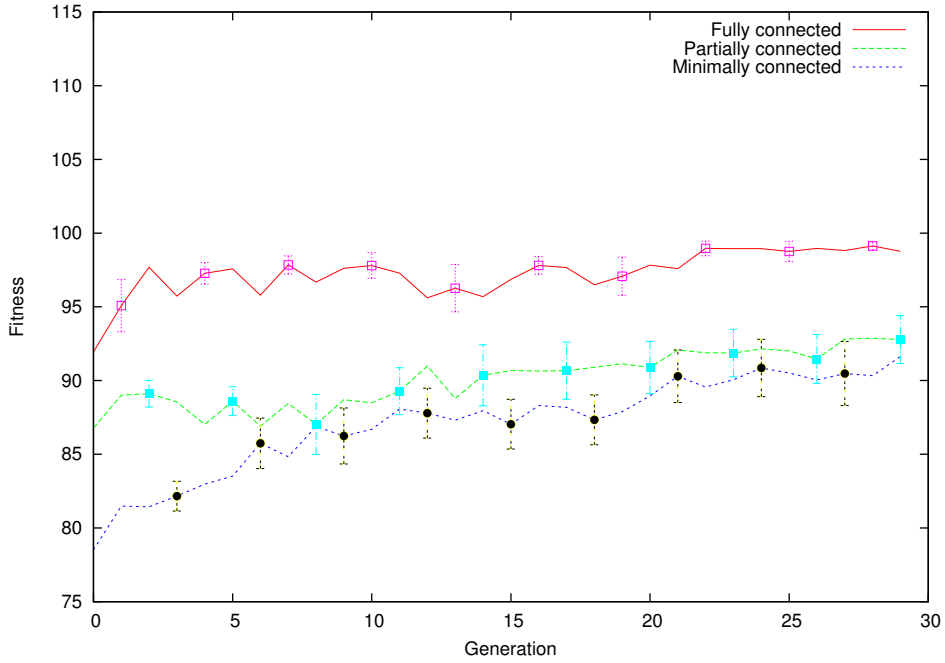

Figure 8: **Impact of initial connectivity on performance of target-reaching controller**. All three controllers used joint velocity control and feedforward networks. Starting with a fully-connected topology performed better for this task. Since in the target-reaching environment robot joints interact with multiple spatial dimensions, starting a fully-connected topology is preferred because it already includes the necessary connections for capturing these interactions.

for the malfunctioning joint.

The leftmost two bars at 0.0 in Figure 9 show the results from this experiment. The two bars compare the performance of the evolved target-reaching controller under normal circumstances and with a joint malfunction. The plot was generated by evaluating the champion network from each evolution trial on 10 episodes. As the figure shows, although there is a reduction in the controller's fitness, the evolved network is still able to complete the task with the faulty joint.

In an attempt to improve the evolved controller's robustness, the evolution process was run again. However, this time, some random negative Gaussian noise was added to the outputs of the neural network before they were passed to the underlying PID controller. The random noise was added on all neural network outputs and was proportional to the magnitude of the network output:

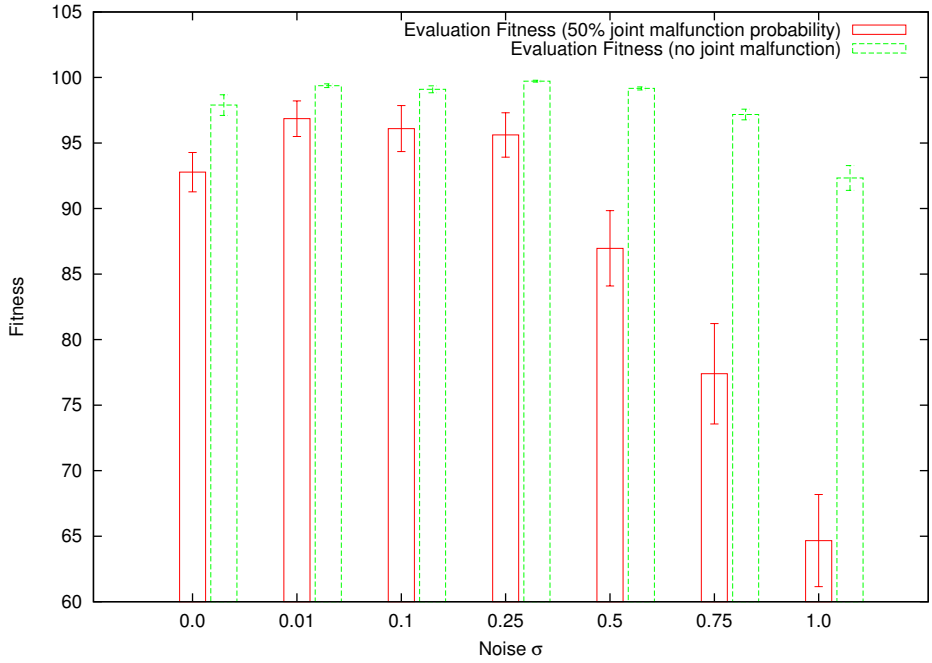

Figure 9: **Impact of actuation noise ratios (σ) on performance of the target-reaching controller under normal circumstances (green dotted bars) and with a malfunctioning joint (solid red bars)**. As the red bars show, small amounts of negative actuation noise improve the controller's robustness against a joint malfunction. The green bars show that the actuation noise improves neuroevolution's performance and reduce the variance even in absence of a joint malfunction.

$$\text{Velocity}_{J_3} = \text{Output}_3 \times (1 - |N(\mu = 0.0, \sigma)|) \tag{4}$$

The solid red bars in Figure 9 compare the impact of various noise ratios on the performance of the evolved controller under joint malfunctions. As the figure shows, adding small amounts of negative noise to the outputs allows the controller to cope with the joint malfunction even better. More specifically, the average fitness for the baseline controller using no noise ($\sigma = 0.0$) was 92.78 with a standard error of 1.50 under a joint malfunction. However, the controllers that experienced small amounts of actuation noise ($\sigma = 0.01$) in training reached an average fitness of 96.86 with a standard error of 1.36 under a joint malfunction. Thereby, the increased robustness is statistically significant.

Interestingly, the added output noise also improves the evolution's results when there is no joint malfunction. To understand why first note that the final evaluation fitness for

an evolved champion network is usually lower than the fitness it receives during evolution. This difference is due to the randomness of the 10 target positions in the environment. Although each individual is evaluated over 10 episodes, for a total of 100 random target positions, still some individuals may receive slightly easier target sets and receive higher fitness scores. Such an individual may not be able to repeat the same performance on another set of 100 targets. For this reason, when the champion network is reevaluated in the end, it might receive a lower fitness compared to the fitness score that made it the champion during evolution.

The dotted green bars in Figure 9 show the impact of adding output noise without a joint malfunction. As the figure shows, adding the output noise can improve the controller's performance and reduce the variance. More specifically, the baseline controllers using no noise ($\sigma = 0.0$) achieved an average fitness of 97.9 with a standard error of 0.79. However, the controllers evolved using a noise ratio of $\sigma = 0.25$ achieved an average fitness of 99.38 with a standard error of 0.14. Thereby, the improved fitness is statistically significant.

The target-reaching experiment shows that it is possible to evolve controllers with precise behaviors for problems with small solution spaces. Moreover, the robustness experiments show that adding actuation noise can improve the controller's robustness against imperfect or malfunctioning actuators. The actuation noise is also beneficial in improving evolution's average performance and reducing the performance variance in absence of malfunctions.

The next section extends the target-reaching experiment to an object insertion task.

3.4 Object Insertion Task

The second task involves inserting a cube that is already grabbed by the suction cup on the end-effector into a cube-shaped cavity slightly larger than the cube. The environment for this experiment is shown in Figure 10. The insertion target is moved to a random position within the robot's reach at the beginning of every episode.

The inputs to the neural network are the (x, y, z) distances from the cube's current position to the center of the cube drop position. The outputs from the neural network control the velocity of the seven joints. The controller is given six seconds at the start of the episode to insert the cube. As before, over the following two seconds samples are taken from the cube's position and its distance to the target location. Controller's fitness is calculated using Equation 2 as was done for the target-reaching task. Every neural network is evaluated for 10 episodes and its fitness is calculated as the average fitness over them.

Figure 11 shows the performance of a controller using direct joint velocity control on this task and compares it with the target-reaching task from Section 3.2. Even though the two tasks have identical fitness functions, the object insertion tasks is more difficult. That is because an effective insertion requires aligning the cube with the hole first before lowering it. An effective insertion of the cube requires some simple planning, while the target-reaching task is entirely reactive.

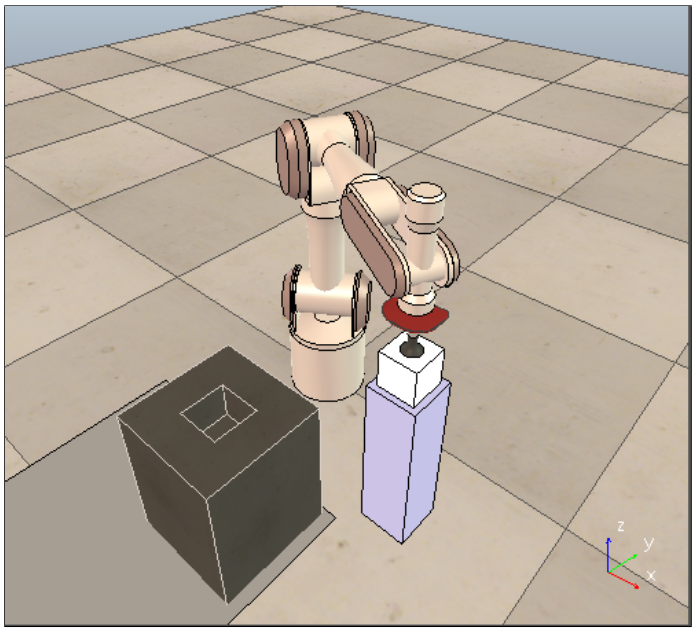

Figure 10: **Setup for the object insertion task**. The episode starts at the configuration shown in the image. The object is already grabbed by the suction cup attached to the end-effector. The robot's objective is to insert the cube into the cube-shaped cavity inside the dark object. At the start of every episode, the insertion target is moved to a random location on the plane underneath it.

The joint-level controller can solve the object insertion task reasonably well. However, it is possible to evolve a better controller by moving to a more high-level controller design where instead of directly controlling the joints the neural network produces signals for an existing IK module. In this setup, the neural network determines the desired position for the end-effector and the IK module together with a standard PID controller perform the command requested by the neural network. Figure 12 shows the overall design of the object insertion controller using an underlying IK module.

Directly passing the output of neural network to the IK module can cause unrealistic or jerky movements, because the network can change its produced target more quickly than what is possible by physical limitations. For this reason, the Reflexxes Motion Library [26] was used to compute online smooth trajectories for the IK target under realistic velocity, acceleration, and jerk constraints. As shown in Figure 13, at every time step the IK module moves the end-effector toward the IK target and the Reflexxes Motion Library module moves the IK target toward the neural network target. Using the Reflexxes Motion Library is not a requirement for using neuroevolution. Rather, it helps with making the

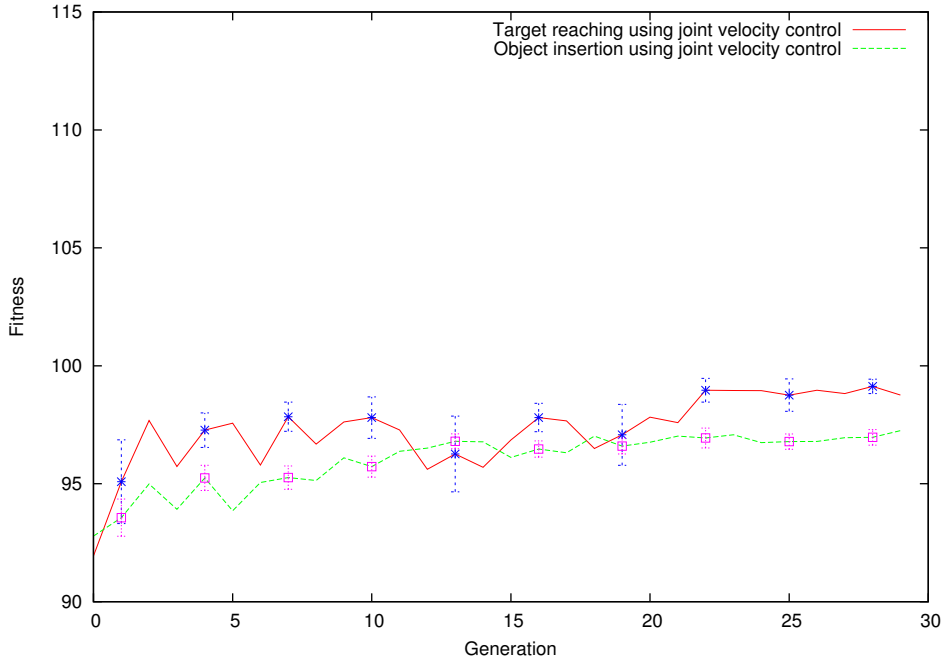

Figure 11: **Comparison of object insertion and target-reaching performance**. Both controllers used velocity control and a feedforward neural network with a fully-connected starting topology. The two tasks use the same fitness function. However the object insertion task is more challenging since it requires aligning the cube first before lowering it into the insertion target.

simulations more realistic.

Conventionally, IK control is done in position mode. That is, as input, the IK controller is given a desired position for the end-effector. However, it is possible to interpret the output of the neural network as a desired velocity for the end-effector instead of a desired position. In this case, the IK module would be used to achieve target velocities for the end-effector instead of target positions. Experiments were done in the object insertion task to compare the performance of IK position control and velocity control. Figure 14 shows the results from these experiments. Using the IK module in velocity mode performs better than position mode. Note that this result is consistent with the result from the target-reaching experiment (Figure 6) where similarly velocity control performed better.

In addition, as Figure 14 shows, the controller using the IK module performs better than the controller using direct joint control. Using the IK module as a lower-level controller allows the neural network to control the robot more easily at a higher level. Also, in this setup, the number of neural network outputs is reduced from seven to three. This reduction

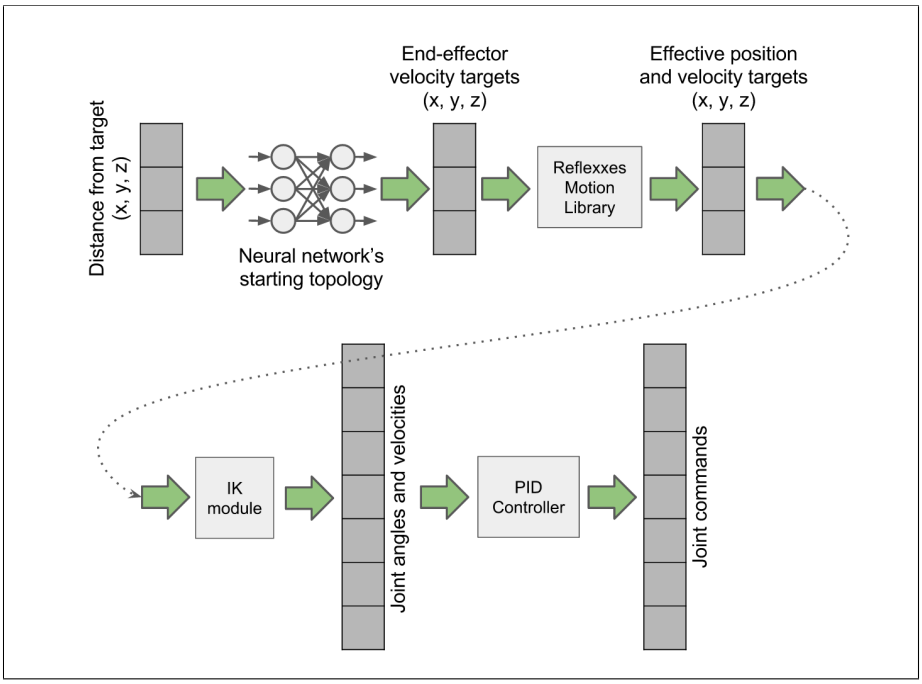

Figure 12: **High-level controller for the object insertion task**. The neural network is given the (x, y, z) distance from the end-effector to the cube as its three inputs. The outputs from the neural network are the (x, y, z) velocities for the end-effector. The velocities are passed to the Reflexxes Motion Library module. This module produces a trajectory to move the end-effector from its current state to the target state. The first point in the trajectory produced by this module is then passed to an IK module, which produces joint commands for a PID controller. This process is repeated at every simulation time step. The Reflexxes Motion Library helps with making the simulation more realistic. The IK module allows the neural network to control the robot at a higher level.

in complexity reduces the number of parameters that need to be optimized by evolution.

In contrast with target-reaching, starting evolution with a partially-connected topology produces better results in object insertion. The reason is that a partially-connected topology connects each (x, y, z) dimension of the input (distance to the cube) to the corresponding (x, y, z) dimension of the output (signals to the IK controller.) Since the behavior of the controller is independent across the three dimensions in this task, a partially-connected starting topology has all the necessary connections. Compared to a fully-connected topology, a partially-connected topology gives evolution fewer parameters to optimize. Figure 15 shows the impact of different choices for the starting topology on this controller.

When using the IK module in position control, the neural network often exhibited an

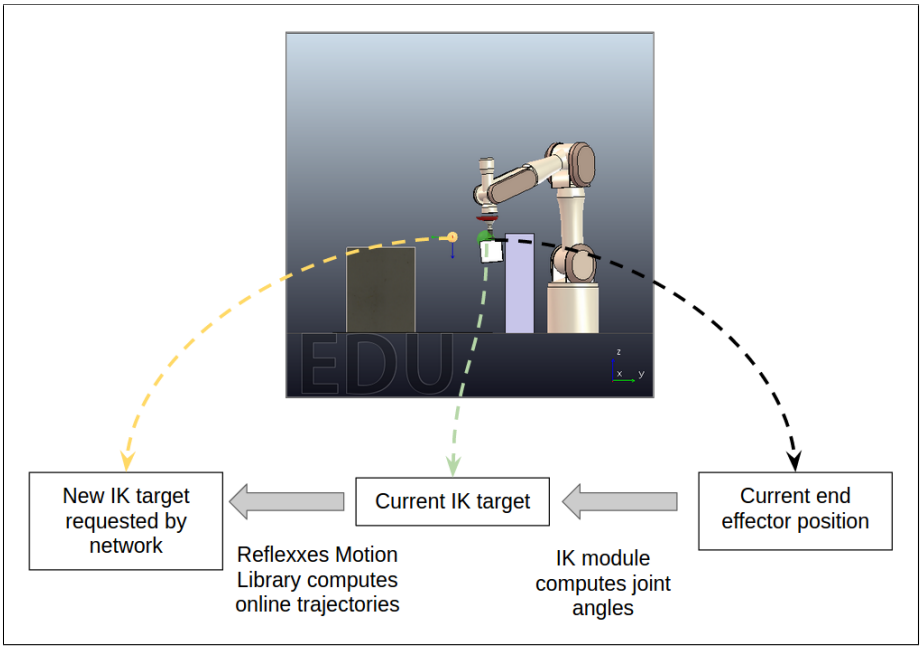

Figure 13: **The end-effector following the neural network target**. The yellow circle shows the IK target produced by the neural network. The green circle shows the effective IK target that is passed to the IK module. The black circle shows the position of the end-effector. At each time step, the IK controller computes the joint angles required to move the end-effector to the effective IK target at the green circle. In turn, the next position of the effective IK target is produced by the Reflexxes Motion Library based on the target requested by the neural network (yellow circle) and the current state of the effective IK target (green circle). This setup allows the neural network to move the end-effector subject to physical velocity, acceleration and jerk constraints.

interesting unintended behavior. It learned to move the end-effector not by generating precise target locations, but by rapidly alternating the network target between two or three points on the limits of the environment's space. An example of this behavior is shown in Figure 16. The controller maintains the cube around the position shown in the image by quickly alternating the network target between the two bright yellow points. A potential explanation for this behavior is that position control is more complicated for the neural network in this task. As the cube approaches its target position, the inputs to the neural network change. A neural network producing positions would need to maintain the same outputs with its changing inputs. So, instead, it has learned to alternative between a few output choices. Indeed, a function mapping a range of different input conditions to a select few output choices requires less complexity than a function producing precise

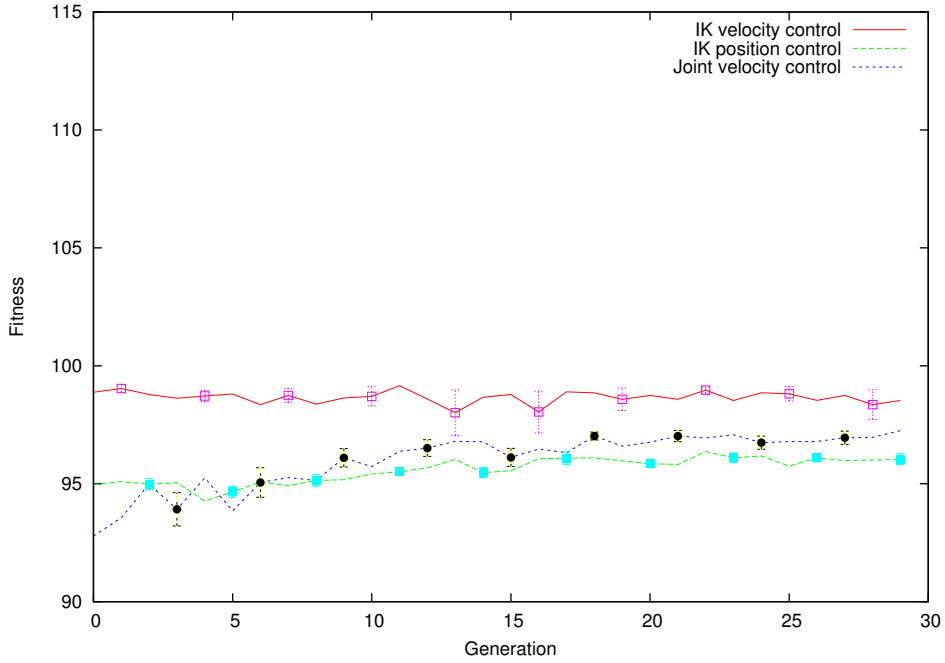

Figure 14: **Comparison of object insertion results using direct joint control with IK control in velocity and position modes**. All three controllers used feedforward networks. The joint-level controller used a fully-connected starting topology. The IK-based controllers used a partially-connected starting topology. IK velocity control performs better than IK position control. Also, IK velocity control performs better than joint velocity control, because the controller using the IK module allows its neural network to control the robot at a higher-level.

real-valued outputs for a wide range of different input conditions. So, evolution is more likely to discover the simpler solution.

The experiments in this section established that neuroevolution is able to discover successful controllers for the object insertion task. The next section discusses extending the object insertion task to include the initial object grabbing as well.

3.5 Object Grabbing and Insertion Task

While in the object insertion task the controller starts with the cube attached to the suction cup, in the third task the controller needs to grab the cube first and then move it to the cube drop position. The neural network is given six inputs: the (x, y, z) distance from the end-effector to the cube, and the (x, y, z) distance from the cube to the cube drop position.

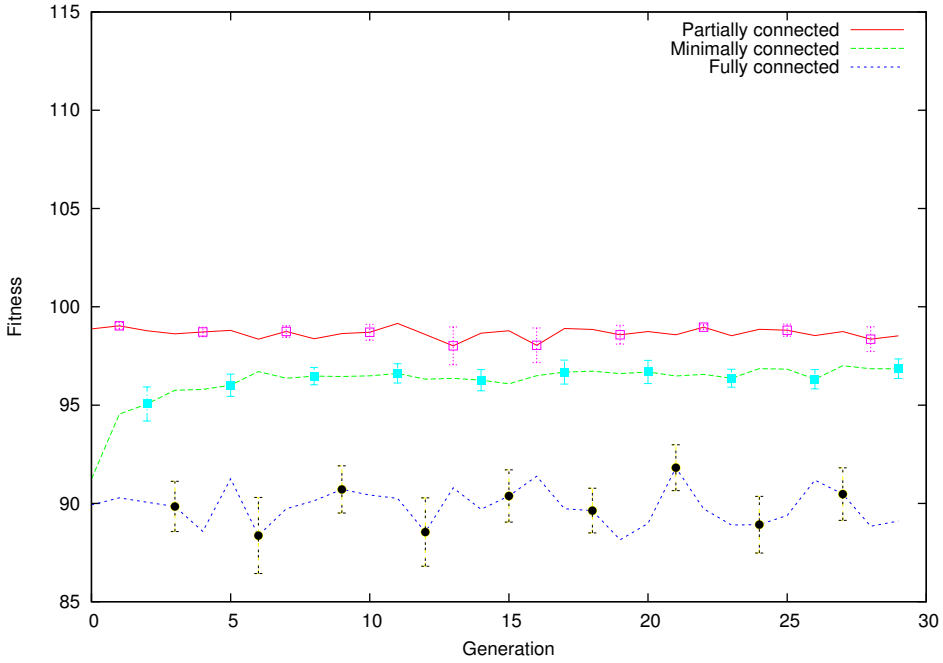

Figure 15: **Impact of initial connectivity on performance of object insertion controller using IK velocity control**. All three controllers used feedforward networks. The neural network in this controller has three inputs and three outputs corresponding to the three spatial dimensions. Since there is little interaction between these dimensions for solving the task, a partially-connected topology has all the necessary connections. Using a fully-connected topology makes the task more challenging as evolution needs to set the unnecessary connection weights to zero. Similarly, using a minimally-connected topology requires evolution to add the useful connections on its own.

The outputs are the (x, y, z) velocities for the end-effector. The velocities are passed to the Reflexxes Motion Library module. This module produces a trajectory to move the end-effector from its current state to the target state. The first point in the trajectory produced by this module is then passed to an IK module, which produces joint commands for a PID controller. This process is repeated at every simulation time step.

As before, the robot is given 10 seconds to complete the task, then for two seconds samples are taken from the cube's position and the end-effector's. The fitness function for this task has two terms corresponding to the two objectives that need to be completed. The fitness equations are

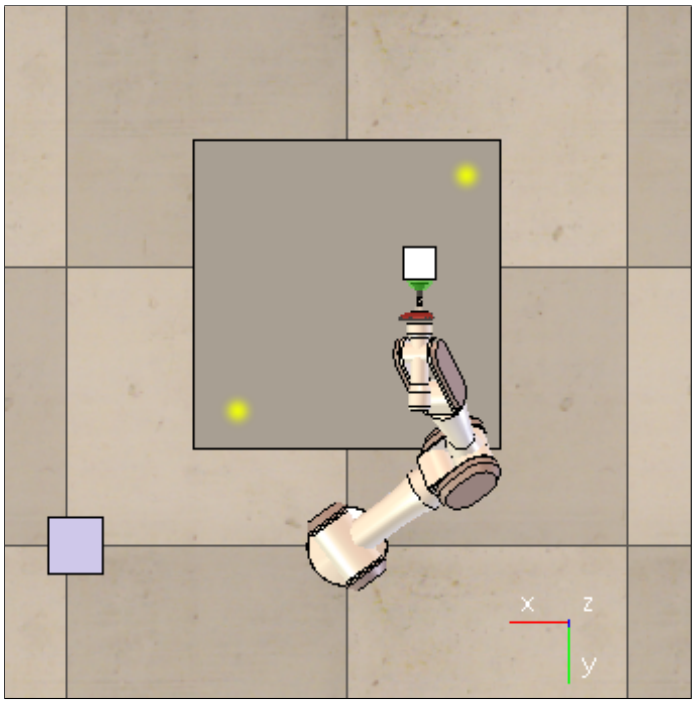

Figure 16: **Alternating targets generated by neural network in IK position control**. The controller maintains the cube around the position shown in the image. Instead of generating an IK target at the desired cube location, it alternates the network target rapidly between the two yellow circles. This observation suggests position control is not optimal for this task. It also suggests that it is easier for neuroevolution to discover a solution based on signaling the direction of movement rather than the target of movement. Such type of control is more in line with velocity control.

$$\text{Fitness}_{\text{grab}} = \frac{50}{1 + \frac{\sum_{s \in \text{Samples}} \text{distance}^2(\text{EndEffector}_s, \text{Cube}_s)}{|\text{Samples}|}} \tag{5}$$

$$\text{Fitness}_{\text{insert}} = \frac{50}{1 + \frac{\sum_{s \in \text{Samples}} \text{distance}^2(\text{Cube}_s, \text{DropPosition}_s)^2}{|\text{Samples}|}} \tag{6}$$

$$\text{Fitness} = \text{Fitness}_{\text{grab}} + \text{Fitness}_{\text{insert}} \tag{7}$$

Every neural network is evaluated for 10 episodes and its fitness is calculated as the average fitness over them.

In this experiment neuroevolution was not able to discover a controller that can complete the task. The behaviors of evolved networks are locally optimizing one of the two fitness terms. Some evolved controllers manage to grab the cube, but cannot move it toward the target. Some other controllers kick the cube toward the drop position and then lower the end-effector close to the dropped cube on the floor. Figure 17 shows the results for this controller. Also this controller has the same setup as the best object insertion controller, there is a considerable drop in fitness compared to the results from the object insertion task in Figure 14.

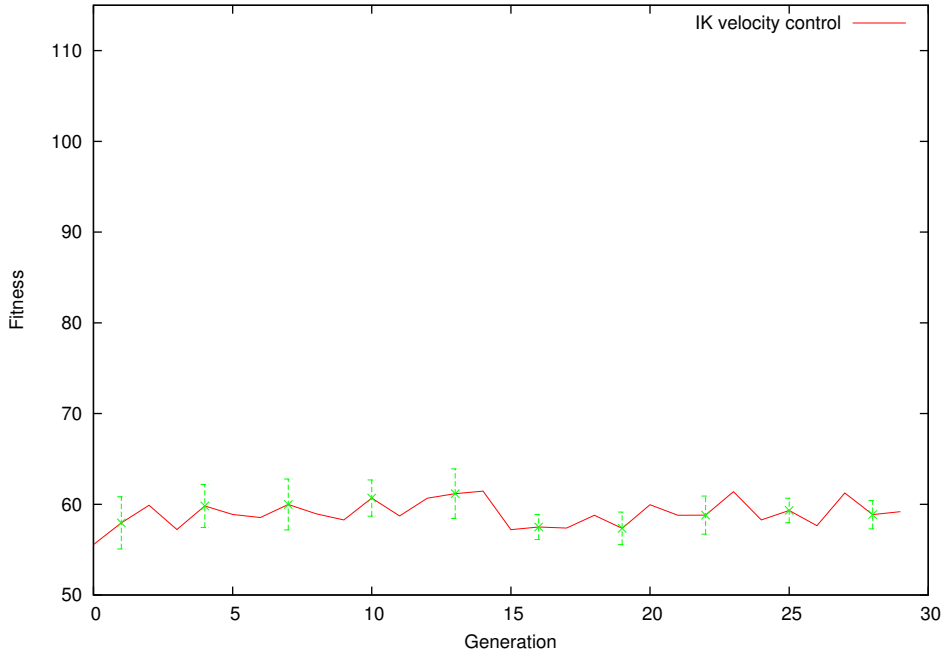

Figure 17: **Object grab and insertion results**. The controller used an IK module in velocity mode. There is a considerable drop in fitness compared to the results from the object insertion task in Figure 14. The object grab and insertion task is challenging for neuroevolution because it requires some planning.

The object grab and insertion task is considerably more challenging for neuroevolution since it requires the robot to fulfill two different objectives. Doing so requires some planning. Even though it is possible to design two disjoint controllers to solve this particular

task in two steps, such a solution will not easily extend to more complex setups requiring more complex behaviors. So, it is desirable to develop evolutionary techniques that can evolve controllers with planning abilities to solve multi-step tasks.

3.6 Conclusions from Completed Work

Results from completed work suggest that neuroevolution is able to discover useful reactionary tactics. Its ability to do so was particularly observable in the neural network that exploited a positional IK controller to signal the desired direction for movement instead of the actual target for the movement. However, results from the object grab and insertion task suggest that it is harder for evolution to discover multi-step strategies and plan for future steps. The next section proposes methods and experiments to address this challenge.

4 Proposed Work

This section proposes to develop a novel neuroevolutionary method that can discover robotic controllers able to complete multi-step tasks requiring planning. The key objective for the method is the ability to discover useful subtasks and a plan without manually specifying the subtasks.

4.1 Controller Design

The proposed controller design is based on a set of Action Sequence Networks (ASNs) and a planning network.

4.1.1 Action Sequence Networks

An ASN operates much like a program subroutine and its purpose is to complete some subtask. Once a higher-level controller activates a given ASN, it takes over the controller and continues to output actions based on the inputs fed to the controller until it terminates. At that point control is returned to the high-level controller which can activate another ASN. Action sequence networks have a similar function as Options in reinforcement learning [42]. Working with continuous inputs and outputs and using neural networks creates different challenges and also different opportunities for automatically discovering effective ASNs without manual input.

Following the Options terminology, an ASN has three components:

1. Policy: The behavior (outputs) that the neural network generates given the inputs.
2. Termination condition: The criteria (observations) for terminating the execution of the action sequence.

3. Triggering condition: The criteria (observations) under which an ASN is usable.

4.1.2 Planning Network

The planning network is the high-level module that triggers and terminates the available ASNs. Figure 18 shows the activation of one ASN by the planning network.

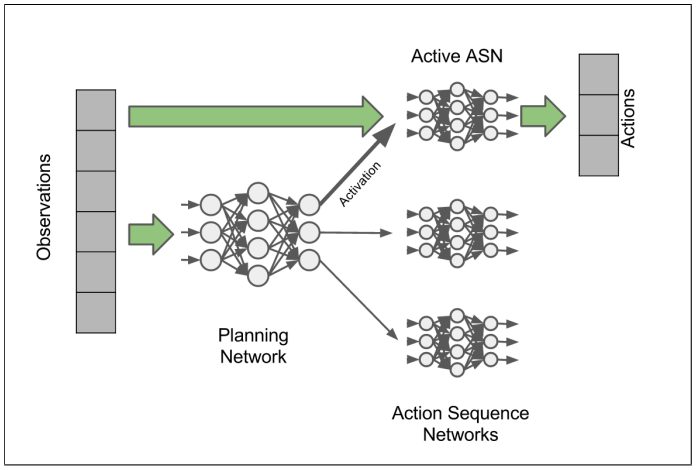

Figure 18: **Activation of an ASN by an input-grounded planning network**. Based on the observations receives from the environment, the planning network can pick one of the three attached ASNs for activation. As long as it stays active, the ASN receives observations from the environment and generates actions to take.

4.1.3 Controller Design Variants

There are multiple ways to wire the ASNs and the planning network together. The overall design of the controller will in turn influence the internal design of the ASNs and the planning network. Three options will be evaluated in proposed work:

Self-Regulating ASNs: In this setup, each ASN has two auxiliary outputs in addition to the task-specific outputs. The first auxiliary output serves to signal the network's preference to be activated. The second auxiliary output signals when the action sequence is complete and should be terminated. With such auxiliary outputs, the planning network can be replaced by a static module that always selects the ASN with the highest activation preference and allows it to control the robot until its termination preference crosses a set threshold value. This setup amounts to a self-regulating ensemble of subtask networks that collectively decide when to activate and

terminate themselves. A controller can be formed by putting together a number of ASNs to form a team.

Input-Grounded Planning Network: In this setup, which is shown in Figure 18, a planning network is used together with a number of ASNs. The planning network has a fixed number of connections for plugging in ASNs. The auxiliary activation and termination outputs are removed from the ASNs and instead the planning network decides which ASN to activate at each time step. The planning network receives the same inputs from the environment as the ASNs, so its decision-making is grounded in the agent's observations.

ASN-Assisted Planning Network: In this setup, the auxiliary activation and termination preference outputs from the ASNs can be fed to the planning network. In this case, the planning network does not receive observations from the agent's environment and makes its decisions based on the preference signals from its connected ASNs.

These variants are designed with the aim of creating controllers that are able to complete multi-step tasks by completing a series of subtasks. The controller design has the necessary components to allow for planning behavior to emerge. However, in order to evolve such controllers, a new evolutionary method is required. The next section discusses the proposed method.

4.2 Evolutionary Planning Method

The proposed evolutionary method is an adaptation of ESP [17]. Since the controller consists of multiple components that need to cooperate to complete the task, evolving separate subpopulations for separate components is a suitable approach. The proposed method is to evolve multiple populations of ASNs together with a population of planning networks.

ESP was originally used to create a single neural network by maintaining a population of neural network fragments with specific roles. The fragments can work together in a team to create a complete neural network. Later experiments with multi-agent extensions of ESP [48] show that in multi-agent environments team behavior can emerge and different agent subpopulations can take on specialized roles. So, it is expected that ESP can be extended to other domains where team behavior is needed.

The proposed method builds on this idea to maintain a population of ASNs and planning networks. A team can be formed by picking a planning network and a number of ASNs. By sharing the fitness between members of a teams, ESP can discover effective individuals as well as effective combinations leading to successful controllers. Also, maintaining separate populations will allow for evolving ASNs with specialized roles.

The goal of this evolutionary method is to discover useful specializations that would align with the subtasks needed to solve the problem. This evolution-discovered task decomposition may or may not match a manual decomposition. The next section discusses proposed experiments to evaluate the proposed method.

4.3 Proposed Experiments

This section discusses the plan for running experiments to evaluate whether the proposed evolutionary method and controller designs are able to solve multi-step tasks. These experiments can show

4.3.1 Object Insertion Using Evolutionary Planning

Although the experiments for object insertion using IK velocity control in Section 3.4 were successful, it will be interesting to evaluate the proposed evolutionary planning method on the same task.

Three experiments will be done to evaluate the performance of the three controller variants discussed in Section 4.1.3. These experiment will show if the proposed method can discover a controller to solve the same task.

Also, since the proposed controller setup is more complicated than a monolithic network, it might require more generations to evolve. These experiment make it possible to compare the behavior of the proposed evolutionary method with the existing evolutionary method used for completed work.

4.3.2 Object Grabbing and Insertion Using Evolutionary Planning

The existing evolutionary method was not able to discover a controller to complete the object grabbing and insertion task in Section 3.5. Three experiments will be done to test the proposed variants of the controller in Section 4.1.3 on this task.

Experiments will be based on using two or three separate subpopulations of ASNs. The expected outcome is for the separate subpopulations to converge to specialized roles that can collaboratively perform the object grabbing and insertion steps.

The ASN activation and terminations of evolved controllers can be studied to analyze whether evolution discovers meaningful roles and subtasks for each ASN or not.

4.3.3 Object Grabbing and Insertion Using Multiobjective Evolutionary Planning

The objective function for the object grabbing and insertion task is based on the weighted average fitness function shown in Equation 7. The ESP-based evolutionary method will be extended to support multiobjective evolution using NSGA-II [4]. In multiobjective evolution, evaluating an individual produces multiple fitness scores corresponding to the

different objectives in the task. Using multiple objective scores can provide more refined fitness information to the selection algorithm. NSGA-II is a selection algorithm that can order the individuals in the population based on their performance on multiple objectives and preserve individuals that perform well on either of the objectives.

Experiments will be done on a multi-objective formulation of the object grabbing and insertion task. In this version, the fitness components calculated in Equations 5, 6 are passed to the selection algorithm as a pair of numbers without being averaged. Experiments will show whether using the refined fitness information can improve the proposed method's performance.

If these experiments are successful, future work can further extend the evolutionary planning method to solve more challenging surrogates for the component insertion task from Section 1.

4.4 Stretch Goal: Sensory-Driven Task Specifications

A important ingredient in an evolutionary process is the fitness function. It is often the case that using a fitness function results in a behavior that successfully maximizes the fitness without resembling the intended behavior that the fitness function was designed to achieve. As an example, when evolving an object grasp and placement behavior on the Atlas robot, evolution discovered a controller that achieved very high fitness by completely skipping the grasping step and instead swinging the arm to hit the object and have it land in the target position by itself!

There are other results that point toward suboptimality of fitness functions for driving evolution. Novelty search [27] is an evolutionary search method that rewards discovery of new and different behaviors without any task-specific objective. Despite not having any measure of closeness to the goal, on some problems it is able to discover solutions that objective-based methods cannot. As another example, results from experiments done on multiobjective evolution [38] suggest that the search process can benefit from periodically disabling stagnated objectives.

These observations serve as motivation to explore the possibility of developing new alternatives for fitness functions. For this purpose, the controller design introduced in Section 4.1.3 can be extended toward developing a sensory-driven task specification framework [5, 12]. Sensory-driven task specification is defined as specifying a goal state by providing only the sensory inputs that the robot would receive from its environment (which may include its proprioceptive sensors) when it has reached the goal state. For example, a robot inserting a cube using visual input is given an image of the situation where the cube has already been inserted from the perspective of its own camera. Alternatively, the robot can be given a sequence of input observations on a path from the current state to the goal state.

The controller design from Section 4.1 is a promising solution for this problem since its decision making is directly grounded in the inputs received from the environment. Consider

a task that is completed by activation of three ASNs. The first two ASNs would terminate when their termination signals are activated by reaching certain states. In order for neuroevolution to complete the task successfully it needs to identify proper transitional states where an ASN terminates and the next one is triggered. On the other hand the ASN's termination signals are computed based on the inputs received from the environment. In other words, the controller's internal coordination is more compatible with sensory-driven task specification than a standard controller using engineered perception and decision making modules. The controller's design can be extended to include extra inputs for ASNs and the planning network to specify a desired target state.

This stretch goal will be attempted if the results from other experiments warrant it and if there is enough time. Otherwise, it will be part of future work that this proposal makes possible.

5 Conclusion

Neuroevolution is a promising tool for developing artificial agents and robotic controllers. In the object insertion domain, completed experiments have fulfilled the requirements for precise behavior and creating controllers that are robust against failures. However, evolving controllers that are able to do planning remains as a challenge. Developing improved evolutionary methods and controller designs will make it possible to apply neuroevolution to realistic control tasks, like the component insertion task, that require multi-step strategies and automatic discovery and completion of subgoals.

References

[1] R. D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509, 1995.

[2] R. D. Beer, H. J. Chiel, and L. S. Sterling. Heterogeneous neural networks for adaptive behavior in dynamic environments. In *Advances in neural information processing systems*, pages 577–585, 1989.

[3] J. Bongard. Evolving modular genetic regulatory networks. In *wcci*, pages 1872–1877. IEEE, 2002.

[4] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. *Lecture notes in computer science*, 1917:849–858, 2000.

[5] K. L. Doty and R. R. Harrison. Sweep strategies for a sensory-driven, behavior-based vacuum cleaning agent. In *AAAI 1993 Fall Symposium Series Instantiating Real-World Agents Research Triangle Park, Raleigh, NC*, pages 1–6, 1993.

[6] T. D'Silva. *Neuroevolution of a Robot Arm Controller*. Computer Science Department, University of Texas at Austin, 2005.

[7] T. D'Silva and R. Miikkulainen. Learning dynamic obstacle avoidance for a robot arm using neuroevolution. *Neural processing letters*, 30(1):59–69, 2009.

[8] D. Floreano, P. Dürr, and C. Mattiussi. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

[9] D. Floreano, P. Husbands, and S. Nolfi. Evolutionary robotics. In *Springer handbook of robotics*, pages 1423–1451. Springer, 2008.

[10] D. Floreano and F. Mondada. Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot. In *Proceedings of the third international conference on Simulation of adaptive behavior: From Animals to Animats 3*, number LIS-CONF-1994-003, pages 421–430. MIT Press, 1994.

[11] M. Freese, S. Singh, F. Ozaki, and N. Matsuhira. Virtual robot experimentation platform v-rep: a versatile 3d robot simulator. In *Simulation, Modeling, and Programming for Autonomous Robots*, pages 51–62. Springer, 2010.

[12] G. Giralt, R. Chatila, and M. Vaisset. An integrated navigation and motion control system for autonomous multisensory mobile robots. In *Autonomous robot vehicles*, pages 420–443. Springer, 1990.

[13] F. Gomez and R. Miikkulainen. Incremental evolution of complex general behavior. *Adaptive Behavior*, 5(3-4):317–342, 1997.

[14] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Efficient non-linear control through neuroevolution. In *Machine Learning: ECML 2006*, pages 654–662. Springer, 2006.

[15] F. Gomez, J. Schmidhuber, and R. Miikkulainen. Accelerated neural evolution through cooperatively coevolved synapses. *The Journal of Machine Learning Research*, 9:937–965, 2008.

[16] F. J. Gomez and R. Miikkulainen. Active guidance for a finless rocket using neuroevolution. In *Genetic and Evolutionary ComputationGECCO 2003*, pages 2084–2095. Springer, 2003.

[17] F. J. Gomez and R. Miikkulainen. *Robust non-linear control through neuroevolution*. Computer Science Department, University of Texas at Austin, 2003.

[18] F. Gruau and D. Whitley. Adding learning to the cellular development of neural networks: Evolution and the baldwin effect. *Evolutionary computation*, 1(3):213–233, 1993.

[19] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[20] G. E. Hinton and S. J. Nowlan. How learning can guide evolution. *Complex systems*, 1(3):495–502, 1987.

[21] P. Husbands, T. Smith, N. Jakobi, and M. O'Shea. Better living through chemistry: Evolving gasnets for robot control. *Connection Science*, 10(3-4):185–210, 1998.

[22] C. Igel. Neuroevolution for reinforcement learning using evolution strategies. In *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, volume 4, pages 2588–2595. IEEE, 2003.

[23] E. M. Izhikevich et al. Simple model of spiking neurons. *IEEE Transactions on neural networks*, 14(6):1569–1572, 2003.

[24] A. Jain, A. Subramoney, and R. Miikulainen. Task decomposition with neuroevolution in extended predator-prey domain. In *Artificial Life*, volume 13, pages 341–348, 2012.

[25] N. Kohl, K. Stanley, R. Miikkulainen, M. Samples, and R. Sherony. Evolving a real-world vehicle warning system. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1681–1688. ACM, 2006.

[26] T. Kröger. Opening the door to new sensor-based robot applicationsthe reflexxes motion libraries. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4. IEEE, 2011.

[27] J. Lehman and K. O. Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *ALIFE*, pages 329–336, 2008.

[28] D. Lessin, D. Fussell, and R. Miikkulainen. Open-ended behavioral complexity for evolved virtual creatures. In *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 335–342. ACM, 2013.

[29] M. A. Lewis, A. H. Fagg, and A. Solidum. Genetic programming approach to the construction of a neural network for control of a walking robot. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 2618–2623. IEEE, 1992.

[30] W. Maass. Networks of spiking neurons: the third generation of neural network models. *Neural networks*, 10(9):1659–1671, 1997.

[31] P. MacAlpine, M. Depinet, and P. Stone. Ut austin villa 2014: Robocup 3d simulation league champion via overlapping layered learning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence (AAAI-15)*, pages 1–7, 2015.

[32] R. Miikkulainen. Neuroevolution. In *Encyclopedia of Machine Learning*, pages 716–720. Springer, 2010.

[33] D. J. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

[34] D. E. Moriarty and R. Miikkulainen. Evolving obstacle avoidance behavior in a robot arm. In *Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, pages 468–475. MIT Press Cambridge, MA, 1996.

[35] D. E. Moriarty and R. Mikkulainen. Efficient reinforcement learning through symbiotic evolution. *Machine learning*, 22(1-3):11–32, 1996.

[36] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[37] J. D. Schaffer, D. Whitley, and L. J. Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 1–37. IEEE, 1992.

[38] J. B. Schrum. *Evolving multimodal behavior through modular multiobjective neuroevolution.* PhD thesis, 2014.

[39] K. Stanley, N. Kohl, R. Sherony, and R. Miikkulainen. Neuroevolution of an automobile crash warning system. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1977–1984. ACM, 2005.

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

[41] K. O. Stanley and R. Miikkulainen. Competitive coevolution through evolutionary complexification. *J. Artif. Intell. Res.(JAIR)*, 21:63–100, 2004.

[42] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

[43] R. A. Téllez, C. Angulo, and D. E. Pardo. Evolving the walking behaviour of a 12 dof quadruped using a distributed neural architecture. In *Biologically Inspired Approaches to Advanced Information Technology*, pages 5–19. Springer, 2006.

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

[45] S. Whiteson and P. Stone. Evolutionary function approximation for reinforcement learning. *The Journal of Machine Learning Research*, 7:877–917, 2006.

[46] D. Whitley, S. Dominic, R. Das, and C. W. Anderson. *Genetic reinforcement learning for neurocontrol problems*. Springer, 1994.

[47] A. P. Wieland. Evolving controls for unstable systems. In *Connectionist models: proceedings of the 1990 summer school*, pages 91–102, 1990.

[48] C. H. Yong and R. Miikkulainen. Cooperative coevolution of multi-agent systems. *University of Texas at Austin, Austin, TX*, 2001.