Punctuated Anytime Learning for Hexapod Gait Generation

Gary B. Parker

Connecticut College, New London, CT, USA, parker@conncoll.edu

Abstract

Punctuated anytime learning is presented as the solution for two problems: the use of anytime learning with an off-line learning module and the linking of the actual robot to its simulation during evolutionary robotics. Two methods of punctuated anytime learning, fitness biasing and the co-evolution of model parameters, are described and compared using the common task of gait generation for a hexapod robot with changing capabilities.

1. Introduction

In order to explore areas of high risk where multiple robots are more suitable than a few robots, it is required that the multiple robots be relatively inexpensive to maintain the same cost. Inexpensive robots with sufficient computational power to maintain autonomous control, yet insufficient to do on-line learning, are plausible; but would not be capable of adapting to changes in the environment or their own capabilities. A highly capable central computer that is safely out of the area of high risk can be used to carry the learning system as long as its learning can be linked to the actual robot. *Punctuated anytime learning* is a way to link the off-line learning system to the robot's on-line controller.

There are two main issues that need to be addressed for off-line learning: anytime learning without immediate feedback and dynamic simulator accuracy in evolutionary robotics. Anytime algorithms were defined by Dean and Bobby [1] to be algorithms that could be interrupted at any time and return a current best answer as they strive to find the optimal answer. Grefenstette and Ramsey [2] used the term *anytime learning* to refer to a method of learning that integrates the execution (control) module and the learning module. This method had the characteristics of anytime algorithms except that the learning model never stopped; it continually gave a meaningful answer any time the solution was improved. A simplified version of the anytime learning system is shown in Figure 1. There are two main modules. The Control Module contains the control program, which directs what actions to take in response to the input sensors. The Learning Module's function is to use an internal simulation of the robot and its environment to learn a better control program. Feedback from the Control Module can be used to update the internal representation in the Learning Module.

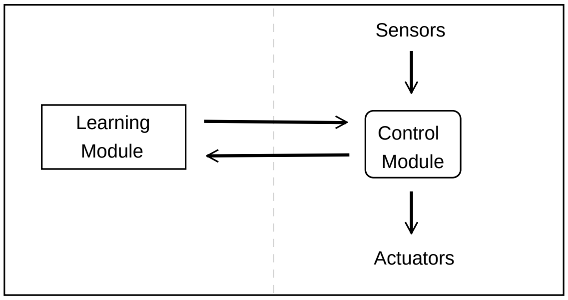

Figure 1: *Typical anytime learning system.*

The anytime learning system was used to solve a simulated problem of cat-and-mouse. The system was to be operating in a mobile agent (the tracker or cat) that was to track a second mobile agent (the target or mouse) that was pre-programmed for avoidance. The anytime learning systems, although they had temporary drops at each phase change, were generally superior to the systems without anytime learning. They allowed a learning component to continually compute a best control solution while the robot operated in the environment using the latest best solution. Grefenstette's system could adapt quickly to changes in the robot or the environment by having the robot's sensors continually update the status of the environment and the robot's capabilities in the learning component's simulation. This is where a difficulty develops when this methodology is applied to off-line learning. For off-line learning, the learning system can only depend on global observation, instead of precise sensors, to determine the robot's capabilities. A modification to anytime learning is required to compensate for the lack of quick feedback, precise sensors, and high computational power in robots.

Evolutionary robotics is a research area that makes use of the various forms of evolutionary computation (EC) to provide a means of learning control systems for robots [4, 9]. Most forms of evolutionary computation require that a population of possible solutions be tested over several iterations. A significant issue of discussion in Evolutionary Robotics is whether or not these tests should be done on simulations. There are three general approaches. One is to do all of the training off line and transfer the results to the actual robot when it is complete [5, 6, 20]. This technique requires that significant attention be paid to the simulation as its accuracy directly affects the results. A model is created to have the key characteristics of the robot. This model can be very complex to better simulate the robot, but this usually means that it will take longer to develop and have increased computation time during learning. A simple model will allow faster computation time, but will not accurately represent the robot. A second method is to do most of the training off line and transfer to the robot for some remaining generations [7, 9]. This requires a less accurate simulation (yet one is still required) and it can take significant time to do the on-line training. The third method does all of the training on line [3, 10, 12]. If the task can be completed and the fitness can be accurately judged in minimal time, this can be a viable option. It precludes the need for any simulation. The problem is that the training takes *num-generations* * *num-individuals* * *time-to-complete-task* time to do the training. An increase in any of these variables results in a multiplicative increase in training time. The problem with all of these techniques is that either time has to be to put into the simulation or into the training on the actual robot. In addition, none of them are designed to be able to refine the solution while the robot is in operation unless it is only doing the task being learned. What is needed is dynamic simulator accuracy, a way of coupling the simulator to the actual robot [4, 8, 11].

This issue was addressed by Nordin et al. [13] as they used genetic programming to learn the best response to a set of stimuli for a Khepera robot. They stored memories of past tests on the robot to use in the evolution of a control function plus a more accurate model of the world. This solution worked well for the reactive control systems they addressed, but could not be applied to the learning of a cycle of activations to control the gait of a hexapod robot.

2. Punctuated Anytime Learning

A solution that provides this simulator/robot coupling and the needed modification to anytime learning is the use of a *punctuated anytime learning* (PAL) system. Training with evolutionary computation takes place offline on a simple model while periodic checks on the actual robot help to improve the learning systems output. This type of anytime learning is referred to as punctuated because the learning, although it is continuous, has break points where it is working on newly updated information about the actual robot. Figure 2 shows a punctuated anytime learning system. In this system, the Learning Module is not part of the robot. Being a separate unit, it no longer has direct input to the Control Module. Updated control programs are sent via some form of external communication. In addition, feedback is no longer sent via a direct link. It can be transmitted, such as the control program, if there is an evaluation unit on the robot. Alternatively, it may be provided to the Learning Module from an outside observer (pictured in Figure 2). The second option becomes more likely as the complexity of the robot goes down. There are currently two methods of punctuated anytime learning: *fitness biasing* and the *co-evolution of model parameters*.

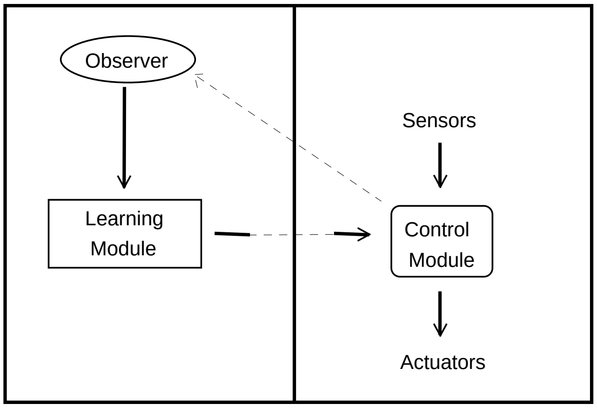

Figure 2: *Punctuated anytime learning system where the learning module cannot be on the robot.*

2.1 Fitness Biasing

Anytime learning with fitness biasing allows the system to modify the workings of the evolutionary computation based on the performance of the robot. The model is not affected; its parameters were set before anytime learning began. The periodic checks on the actual robot alter the processing within the evolutionary computation through the biasing of evolved solution fitnesses by comparing their performance on the model to their performance on the actual robot.

Probability for selection is determined by computing each individual's fitness on the robot model. This fitness is computed for each individual in the population. After each *n* generations all solutions of the EC population are tested on the actual robot. These measurements are used to bias the fitnesses found on the model to make them equal to the actual robot fitnesses. These biases are used as the EC continues training. In this way, solutions that are good on the actual robot but poor on the model have boosted fitnesses during training, which results in their production of more offspring.

This solution requires *population-size* actual tests every *n* generations.

```
Model-Fitness = Compute-Model-Fitness(Solution)
if absolute-value(Model-Fitness) < 1
   Bias = 1
else
   Actual-Fitness = Test-on-Robot(Solution)
   Bias = Actual-Fitness / Model-Fitness
```

Figure 3: *Algorithm to compute bias.*

A bias for each solution of the population is computed using the algorithm in Figure 3. This bias is stored with its corresponding solution. It is used in subsequent generations of the EC to alter the fitness of the solution computed on the model of the robot. . This is done by multiplying the fitness computed on the model by the bias:

$$\text{Corrected-Fitness} = \text{Model-Fitness} * \text{Bias} \qquad (1)$$

These *Corrected-Fitness*es are used for selection during the subsequent training being done by the EC. Pairs of individuals are stochastically selected for reproduction using the *Corrected-Fitness*es. The two individuals produce a single offspring for the next generation; combining their attributes by crossover with possible random variations caused by mutation. The new offspring's bias is computed by averaging the biases of its parents.

2.2 Co-Evolution Of Model Parameters

The co-evolution of model parameters dynamically links the model to the actual robot. Periodic tests of evolved solutions on the actual robot are used to co-evolve the accuracy of the robot's model with the EC produced solution. Every *n* generations of training on the model, three solutions (best, random, and worst (or another random)) are tested on the actual robot. Each of these three is run on the actual robot to gain a true fitness. This fitness should be general enough so that it can be acquired by external observation (such as an overhead camera). The three gait solutions are tested on each of the individuals in the model population. A comparison of each gait solution's fitness on the actual robot verses the model is used to determine the fitness of each individual of the model population (Figure 4).

Each individual is made up of a set number of genes. Each gene represents a corresponding field in the robot's model. These genes evolve to produce models that correspond in performance to the actual robot. A comparison of each gait solution's fitness on the actual robot verses the model is used to determine the fitness of each individual of the model population. A model that performs similarly to the robot on all three tests is assumed to have accurate parameters and is given a high fitness. The population of model parameters is started out through random generation. Alternatively, individuals can be generated by producing perturbations of the original measured parameters. After the model parameters are checked at the *n*th generation, the best is used to continue evolving gait solutions. The fitnesses applied to the individuals in the population of model parameters are used in the co-evolution of the model parameters, which will continue to evolve until interrupted after another *n* generations of gait learning.

```
R1 = Test best solution on the robot
R2 = Test random solution on the robot
R3 = Test worst solution on the robot
For each individual in the model population
    M1 = Get model fitness using best solution
    M2 = Get model fitness using random solution
    M3 = Get model fitness using worst solution
    Order-Same = True if the fitness order of R1, R2, &
                 R3  is the same as M1, M2, & M3
    Signs-Same = True if Rx & Mx have the same sign
                 for x = 1 to 3
    Fit-Comp = Fitness of the robot to model comparison
    If neither Order-Same or Sign-Same are True
          Return  .001
    If either Order-Same or Sign-Same are True
          Return  .001 x Fit_Comp
    If both are True
          Return Fit_Comp
```

Figure 4: *Algorithm to compute the fitness of each model's set of parameters in a population.*

For some problems, a genetic algorithm will not be the best optimization technique for finding the model parameters. It may be hill climbing or any other form of greedy algorithm. While the CGA is learning the cycle for the gaits, or whatever else is being learned, the model parameters can be learned at the same time, whether it be by some form of evolutionary computation or through some other means of learning. In either case, the concept of co-learning model parameters is the same.

3. Using PAL to Learn Gaits

Learning gaits that are adaptable in changing environments and with variable robot capabilities is of importance in the actual implementation of autonomous legged robots. A satisfactory gait requires that the proper sequence of commands are given to the various actuators of the robot in an order that results in the coordinated movement of each leg, plus the coordinated movement of the set of legs. Learning must be done continuously with the best possible gait to increase the effectiveness of the robot. Both forms of PAL have been used to adapt evolved gaits to changes in the robot's capabilities. The co-evolution of model parameters has been tested in simulation [17] and on the actual robot [19]. Fitness biasing, which requires a greater number of tests on the robot, has been tested in simulation only [16, 18]. Tests on the actual robot will be done following the completion of a training environment.

3.1 The Robot

The ServoBot was developed by David Braun to facilitate legged robot control experimentation. It is an inexpensive hexapod robot that has two degrees of freedom per leg. Its body is made of pressed wood fiber board that has been cut into the shapes of leg segments and structural support pieces. Twelve hobby servos, two per leg, provide thrust and vertical movement.

The controller, a BASIC Stamp II, can produce a pulse signal that drives the servos. An input to the robot controller is a twelve bit number (an *activation*) where each bit represents an actuator. A signal of 1 moves the leg back if it is a horizontal actuator and up if it is a vertical actuator. A signal of 0 moves it in the opposite direction. A control sequence is transmitted to a controller through lines that connect to a workstation. Once the control sequence is transmitted, the connection can be interrupted allowing autonomous operations. The controller can store and execute the sequence of primitive instructions downloaded. Some section of the sequence of instructions can be designated for repetition and the controller will continually loop through these instructions.

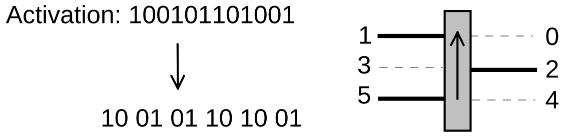

Figure 5: *Results of an activation on the robot.*

An activation (Figure 5) can be thought of as six pairs of actuations. Each pair is for a single leg with the first