Transformation of robot model to facilitate optimization of locomotion

Nick Collins
Department of Computer Science
The University of Texas at Austin
no1uno@cs.utexas.edu

Abstract. Locomotion algorithms for robots generally involve a large number of tuned parameters which are highly sensitive to the shape, size, and mass of the robot. Optimal parameters for one physical model may render another completely immobile. An optimization algorithm can be used to find optimal parameters for a given model, but often requires the user to guess, via trial and error, a viable set of parameters to start with. Since robot locomotion is so fragile, searching for a set of parameters that is at all stable can be very time consuming. This paper proposes a method to tune parameters for a new model if viable parameters are known for a previous model. This is accomplished by slowly transforming the original model into the new model over the course of the optimization process.

1 Introduction

Engineering effective locomotion routines for a humanoid robot is a difficult task. At the very least, robots must be able to stably stand upright, walk forwards and backwards (and perhaps in other directions), and get back up after a fall. The noisy real-time domain, high degrees of freedom, and lack of inherent stability ensure that neither a purely analytic mathematical approach nor a purely experimental policy search will succeed. As such, humans typically develop a walking technique that is complete up to the specification of finitely many real-valued parameters. Mathematically deriving an optimal set of values, or policy, for these parameters is infeasible, so an optimization algorithm is applied. The optimizer is generally seeded with an initial set of policies, and then each generation it picks new policies based on which of the old policies were successful. In a locomotion task, a robot that cannot move at all or is highly unstable achieves minimal fitness. Since locomotion requires very tight coordination between various limbs, nearby policies are likely to be incompetent as well. Therefore, the fitness landscape around any arbitrarily chosen policy is likely to be flat - no matter what nearby policy the optimizer chooses, the robot will be equally unable to make any progress at all, and no useful information about the fitness gradient will be obtained. Therefore, the optimizer must be seeded with policies that are fairly stable and are able to make measurable progress. However, finding policies that are sufficiently stable is part of the problem. Since parameters are likely to be sensitive to the robot’s shape, a stable policy for one model might be useless for
a different model. Nonetheless, parameters that are stable for a certain model are likely stable for similar models. If a stable policy is known for one model, and if that model is used in the first generation but subsequent generations change the model slightly to make it more like the new model, then the optimizer should at any point have a stable set of policies to work with. It should at every point make progress until it has finally learned a good policy for the new model. Thus any sufficiently similar new model could be optimized hands-free without the laborious task of manually discovering a new stable policy.

This task is particularly applicable to the 3D simulator competition in Robocup. This domain captures many of the real-world physical difficulties in designing bipedal robot software while allowing for rapid development and frequent testing. Currently, the competition involves a fixed robot model based roughly on the Aldebaran Nao [1] used in the standard Robocup competition. However, because simulated robots do not need to be manufactured, it’s possible that the simulation competitions could involve robots of varying sizes and shapes. This idea has been applied in the 2D simulation league, producing research in strengths and weaknesses of various designs. Extending to the 3D realm will help bridge the gap to potential heterogenous robots in the real world. The ultimate goal of RoboCup is to produce a team of robots that can compete against the 2050 World Cup champions [2]. In such a competition, it would be important to have robots of different models to exploit different strengths and to mitigate individual weaknesses, just as human players are not all the same size and do not all have the same strengths and weaknesses.

In order for the robot to learn to walk, it must be able to get up successfully after a fall. At time of writing, the UT get-up routine was hand tuned via extensive trial and error to achieve a passable but overly cautious solution. Thus, the contributions of this paper are

1. to have a program optimize a fast and stable get-up routine using the sub-optimal hand tuned solution as a starting point, and
2. to have a program optimize a fast and stable get-up routine for a novel model using the hand tuned solution to the original model as a starting point, freeing humans from having to develop a new starting point.

2 Domain Description

The 3D simulator is a physically detailed simulator that is used for a Robocup competition that is separate from but similar to the real-world Robocup soccer competition. The competition is held annually, with nearly thirty teams from eleven countries on five continents participating in the 2011 competition [2]. The simulator emulates a real-world soccer field in three dimensions with a physics engine that handles collisions, friction, and gravity in a realistic manner. The simulated robots are rough approximations to the real world Nao robot. They are humanoid, having two arms, two legs, a neck and a head, with a total of 22 degrees of freedom. They stand 57 cm tall and have a mass of 4.5 kg [3]. The robot is controlled by a client program that sends torque commands to
the simulator every 20 ms. The robot has access to noiseless information about the current angles of its joints. The UT-AustinVilla code computes desired joint angles for various joints and uses a PID controller to determine the torque to send to the simulator in order to achieve those joint angles. Aside from the lack of a flexible spine, the robot’s body structure resembles that of a human - it has essentially the same limbs and joints as a human, with each having comparable dimensions and range of motion. Therefore, the simulated robot faces very similar locomotive challenges to those encountered by a human infant. In order to perform a particular task, such as getting up from a prone position, the robot must choose various joint angles at particular times. The robot must be able to generate enough upward force to get up but not so much that it jumps, and must be able to get into a balanced, neutral position upon standing.

Fig. 1. The limbs and joints of the simulated Nao model.
3 Related Work

In 2006 Josh Bongard, Victor Zykov, and Hod Lipson developed "self-aware" robots - robots capable of using experimentation to learn their own body structure from an initial state of ignorance [7]. Part of the motivation was to create robots capable of adapting their ability to walk after injury. A limb would be removed and the robot would recognize the change and reform its model so that it could continue to walk. The authors incorporate the ability to develop new locomotion routines following spontaneous debilitation. The work of this paper could complement that work as an alternate method for redeveloping locomotion after a change in model - once the robot has discovered and appropriately modeled the change, it could develop a new locomotion routine by simulating a transformation from its current form to its new form while applying a learning algorithm. Future work would explore if this strategy is more efficient than the one used by Bongard et. al.

Much work has been done on reinforcement learning in non-stationary environments, but in most such work the changes in the environment are regarded as an external force that must be contended with and the goal is to develop techniques to overcome the obstacle presented by a non-static and uncertain environment. This work, on the other hand, always involves environments that are known, in the sense that an accurate model of the robot’s form is available to the agent and the optimization algorithm at all times. Rather than developing learning techniques to contend with challenges presented by external but gradual change in the environment, this approach develops a technique of changing the environment in a gradual but deliberate manner to contend with drastic but known environmental differences. Research of this nature is difficult to find, likely due to the great rarity of a known environment. In most reinforcement learning tasks, the nature of the environment is unknown and cannot be modified by the optimizer. Thus, there would be little interest in studying a situation where the environment can be molded as a tool to enhance optimization. This situation, however, is clearly the case in the domain of differential simulated robot models. Thus, more research is needed in the area.

4 The Get Up Routine

In order to get up from a prone position, the robot must choose certain joint angles at certain times to control the movement of its body. The UT Robocup team devised a routine which iterates through a series of poses, transitioning from one pose to another after a pre-specified period of time. The poses are determined by the joint angles required, with each joint angle being applied symmetrically to both legs. Thus, the get up routine can be numerically parameterized by a sequence of time intervals and a set of joint angles. Originally, these joint angles and time intervals were chosen manually by a human engineer via trial and error. In this paper, the joint angles are chosen by a learning algorithm. Every get up strategy uses the same basic structure, and a particular pose always involves a particular set of joints, but the precise angles to set those joints at is optimized.
The robot detects that it is not upright when its accelerometers indicate that the force of gravity is pulling in a direction not parallel to its torso. When this happens, the robot spreads its arms outward to its side at 90 degree angles. This way, when the robot lands it will fall on either its back or its front. After extending its arms to make sure it falls on its front or its back, the robot pauses for 0.7 seconds before determining which way it has fallen. This paper involves getting a robot to get up from lying on its back. The "get up from back routine", which will be referred to below simply as the "get up routine", the robot iterates through the series of poses shown in Table 1. Figures 2 thru 6 illustrate this for the original hand-tuned get up routine. For each of the poses, the robot sets the target angles specified and then waits the specified period of time before moving on to the next pose. However, any target angle set in one pose is preserved to the next one. So if Pose 2 indicates that the thigh joint should be set to 90 degrees, and then it waits 0.01 seconds before beginning the next pose, the thigh joint will continue moving towards 90 degrees, only stopping when it reaches that target or a new target is set. In other words, the poses only indicate what targets will be set and when they will be set - joints are never "reset" to default positions and movement towards a target does not end at the beginning of the next pose. The "Original X" columns give the values used in the hand-tuned routine, which are used to seed the optimization procedure. Joint angles that were set to 0 in the original hand-tuned code were not optimized.

**Fig. 2.** The robot begins the get up lying on its back.

**Fig. 3.** The robot propels itself up with its arms.

**Fig. 4.** The robot throws its arms forward and contracts its legs to get its center of mass in front of its feet.
Fig. 5. Using momentum from the initial push the robot has managed to roll into a squatting position.

Fig. 6. The robot can get up from the squat by extending its knees and hip.

Table 1. Joint angle targets and wait times of get up routine. The joint names are defined in Figure 1.

<table>
<thead>
<tr>
<th>Pose</th>
<th>Original Wait (s)</th>
<th>Joint</th>
<th>Original Value (degrees)</th>
<th>Optimized?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
<td>Arm 1</td>
<td>-120</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Arm 2</td>
<td>35</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Arm 4</td>
<td>60</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 3</td>
<td>15</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>0.04</td>
<td>Leg 3</td>
<td>110</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>0.2</td>
<td>Arm 1</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Arm 2</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 1</td>
<td>-35</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>0.2</td>
<td>Arm 1</td>
<td>30</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 1</td>
<td>-80</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 4</td>
<td>-60</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 5</td>
<td>-70</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 6</td>
<td>-40</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>0.4</td>
<td>Arm 1</td>
<td>30</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 1</td>
<td>-60</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 5</td>
<td>-80</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>N/A</td>
<td>Arm 1</td>
<td>-90</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 1</td>
<td>-30</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 3</td>
<td>30</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 4</td>
<td>-10</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Leg 5</td>
<td>-10</td>
<td>Yes</td>
</tr>
</tbody>
</table>
5 Optimization of the Get Up Routine

5.1 Fitness Function

To determine the robot's ability to get up, a fitness function must be designed. This function evaluates the robot's abilities and produces a real-valued score, with higher scores representing better performance. A robot that is capable of standing can easily be given a continuous, meaningful fitness based on how long it takes to get up or its stability once standing (as measured by amount of extraneous movement or likelihood of falling back down). However, judging the quality of robots that are not actually capable of successfully standing up is very difficult. Such a robot would have to be rewarded according to how "close" its behavior was to properly getting up, or it would have no information with which to learn. Simple criteria - such as rewarding the robot for raising its center of mass or bringing its center of mass between its feet - can be used. However, they are prone to reward simple but counterproductive behavior while ignoring complicated but useful behavior. For example, the robot can arch its back or try to launch itself upwards in order to raise its center of mass, and it could raise its legs over its torso to bring its center of mass between its feet. Neither criteria is likely to encourage the robot to bend its knees, which is an essential component of how humans and the hand-tuned routine get up.

The purpose of transforming the robot is to obviate the need for a fitness function that can successfully judge incompetent policies. A valid policy for the original robot exists, and if the transformation is sufficiently gradual, some policies that are valid for one model should also work for the next one. Thus, the fitness function used does not attempt to compare incompetent models, instead giving them all very low fitness that is well below that received by viable models. A simple function satisfies these criteria.

The function essentially evaluates the robot by forcing it to fall and seeing how long it takes the robot to get up. Each evaluation begins with the robot being set into an upright, neutral stance by the simulator. The robot is given 1 second to make sure it is stable, and then it is forced to fall backwards. The robot uses accelerometer readings to determine whether it is in an upright position. If it detects that it is not, it will set an internal flag 'hasFallen' and begin the get up routine described above. Once the routine completes, it checks to see if it is upright, and if it is, it clears the flag. Otherwise it continues trying the get up routine. During the evaluation, the robot records how much time the flag 'hasFallen' spends being true, and its fitness for the run is the negation of that. So if it falls once, gets up after 3 seconds, and stays up, its fitness is -3. After forcing the robot to fall, the evaluation runs for 5 seconds and then ends. If the robot gets up in 2 seconds, but is unstable and falls back down after being up for a second, then its fitness will be -4, since the total time it spent falling or getting up was 4 seconds. Punishing subsequent falls serves to make sure the get up routine is stable. If the robot never gets up, it will score nearly -5. There are small errors in calculation of the time spent fallen or getting up, and the optimization algorithm can exploit this by setting all the wait times to 0. Thus,
an optimized invalid policy can score above -4, but they don’t get much above -3.5. For comparison, the hand tuned policy consistently gets scores of around -2.5. Thus, the scores clearly differentiate valid solutions from invalid ones. For each trial, the evaluation is run 15 times and the scores are averaged to compute that trial’s fitness value.

5.2 CMA-ES

The learning algorithm used is Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [6]. CMA-ES progresses in generations, with each generation sampling policies to be evaluated from a Gaussian distribution over the parameter space. The distribution in any given generation is centered in an area of the parameter space that is expected to have high fitness, and its covariance matrix is such that it is expected to sample policies with high variation in fitness. The initial Gaussian distribution is determined by a seed provided by the human running the algorithm. The seed includes an initial policy, which is used as the initial mean value, and a set of standard deviations, which are used for the initial covariance matrix (the non-diagonal entries of the first covariance matrix are set to 0). At the beginning of each generation, a set of policies is sampled from the current distribution. Each item in the sample is evaluated for fitness. The fitness values are ordered and each sample item is given a weight based on its rank in the ordering. Only the rank in the ordering is taken into account, so the amount of difference between fitness values is irrelevant. Hence, a monotonically increasing transformation of the fitness function has no effect on the results of the optimization. The new distribution mean and covariance matrix are chosen so that points near the best policies have a high chance of being chosen in the next generation, and so that variation in the distribution approximates the variation in fitness landscape. There are also two evolution paths, each stored as a vector, which incorporate multigenerational memory and influence the new mean and covariance. CMA-ES keeps track of a step size, which determines how far away from the preferred mean samples can come from. One of the evolution paths serves to alter the size of the step so that each change of mean is expected to be in a direction perpendicular from the previous one. This serves to make sure that the algorithm does not overstep while also making it efficient by making as much progress as possible each generation. The other evolution path keeps information on the effectiveness of previous search steps and directly influences the location of the new mean position and the new covariance matrix. If the mean travels in a general direction over a few consecutive generations, this path will point in that direction and push the mean in that direction in subsequent generations. Thus, both evolution path vectors get longer if the mean moves in the same direction for multiple consecutive generations.

CMA-ES is used because it has proven highly successful for previous optimizations in the 3D Simulator. Namely, it is the optimization algorithm used by the UTAustinVilla team to optimize the walk for their entry in the 2011 3D simulator competition. This agent won the tournament without having had a single goal scored against it in any of the games [4]. In a competition where the
quality of the walking algorithm nearly dominates all other aspects of design, this is strong evidence of the algorithm’s quality. Furthermore, in 2010, Urieli et. all showed that for optimizing the robot’s forward walk, CMA-ES performed significantly better than the cross-entropy method and a genetic algorithm [5]. While walking forward is significantly different from getting up after a fall, they are both tasks within the general domain of locomotion in the described 3D simulator. Many of the problems, including choosing appropriate joint angles to coordinate complex real-time motion, is shared between the tasks, so it is reasonable to expect significant correlation in quality of optimization algorithms between the two tasks. Further, a key attribute of CMA-ES is that it only takes into account the ordering of the fitness landscape. This means that a fitness function need merely correctly distinguish which policies are better and not by how much. For reasons discussed earlier, developing a fitness function that appropriately encodes relative quality information and provides continuous fitness gradients for different policies can be rather difficult. Thus, it is highly useful for optimizing a get up routine to have a learning algorithm that is not sensitive to fitness gradients and is relatively indifferent to the continuity or other differential properties of the function.

The use of other learning algorithms for this task is outside the scope of this paper, but is a good prospect for future work.

5.3 Optimization of Original Nao

In a preliminary test to see what could be expected of get up routine optimization, the original Nao model was optimized using the hand-tuned policy as a seed. The initial standard deviations were set to 10 degrees for angle parameters, and 0.1 seconds for interval parameters. Such values were chosen to strike a balance between having enough variation in the initial populations to prevent premature convergence to a local optima, and staying close enough to the viable solution to prevent immediate divergence. The sample size (i.e., the number of policies to evaluate each generation) was set to 150. The results of the optimization are shown in Figure 7. The x-axis is the generation number, and the y-axis is the average fitness of all policies sampled by CMA-ES in that generation.

Because the get up routine produced by this optimization was to be incorporated into actual RoboCup gameplay, the evaluation function only did 7 runs per trial, and at the end of each run the robot was required to walk in a certain direction immediately after standing in order to test its stability. This aspect was not carried over to the transforming optimizations because the target model has not been optimized for walking and is fairly poor at it. There are also other small differences in the two fitness functions. Therefore, the fitness values of this optimization are not directly comparable to those of the transforming optimizations discussed later. Nonetheless, the general trend in the graph is the same. Figure 7 shows that the optimization significantly improved the average quality of the get up, mainly by improving its speed. The optimized routine, along with an optimized "get up from front routine", were incorporated into the
current UTAustinVilla agent to test whether the improvement had an actual effect on game performance. Figure 8 shows the results of running UTAustinVilla with optimized get up against a variety of high quality opponents from the 2011 RoboCup competition, compared with running UTAustinVilla with the original hand coded get up against the same opponents. Each UTAustinVilla team played 100 games against each opponent, and the average goal differential (number of goals scored by UTAustinVilla minus number of goals scored by opponent) and standard error were computed. Although the difference was only statistically significant for two matchups, the optimized version did better in every single one. This shows that an optimized get up is a valuable aspect of a potential RoboCup team.
6 Transforming Optimization

The method of transforming the robot during optimization was tested by producing a disproportionate variant of the simulated Nao robot and performing optimizations with various transformation schedules.

6.1 Target Model

Care must be taken when developing a target model for which to optimize the get up routine. For many models it is likely that there is no set of parameters that can allow the robot to successfully get up. For example, since the get up routine relies on the use of arms to push the robot up and change its center of mass, it may be impossible for robots with very short or light arms to get up. Knowing in advance whether there even exists a get up routine for a given model is difficult and would require methods outside the scope of this paper. On the other hand, the target model must be sufficiently different from the original that the parameters that worked for the original are useless for the target model. A target model was developed that met both requirements. Figures 9 and 10, representing the original and target Nao respectively, are shown below for comparison. The only difference is that while the original has an upper leg 0.12 m long from hip to knee and a lower leg 0.1 m long from knee to ankle, the target has an upper leg that is 0.09 m long and a lower leg that is 0.055 m long. The limb densities are kept the same, so the masses differ proportionately. This accounts for a 25% and 45% decrease in the length of each limb, respectively, and a 7.4% decrease in overall mass. Not only is this a large change in an important area of the robot, it changes both the proportions of the leg parts to the body and the leg parts to each other. Such a disproportionate change presents a considerable challenge to a learning algorithm or human engineer.
Fig. 9. The original simulator Nao
Fig. 10. The target model

Nao robot model (All values in mm)

<table>
<thead>
<tr>
<th>Pitch</th>
<th>Yaw</th>
<th>Roll</th>
<th>YawPitch</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Total Mass = 4,270 g
6.2 Transformation Schedules

To do transformation during optimization, the two tasks must somehow be interleaved. The algorithm must decide when to transform the current model a little bit, bringing it closer to the target but probably decreasing its fitness, and when to run an optimization generation on the current model, improving its fitness but potentially overlearning. The more gradual the transformation is at each step, the slower the algorithm is, but the less likely it is that the new model will be incompatible with the current policy distribution and be unable to get up. Optimizing on the current model can be either a benefit or a curse - if the target model must use a significantly different approach to getting up than the current one, doing much optimization at the current position will overlearn and make it hard if not impossible to eventually find a workable solution for the target. On the other hand, optimizing can cause CMA-ES to converge on an area of the policy space that is highly stable. When this happens CMA-ES will more consistently sample high quality policies that tend to be more robust to changes in the model.

One technique that is used to improve stability is to optimize the current model until a plateau criterion has been met. The principle behind this is to optimize until a highly optimal solution is developed, thus promoting stability and robustness, at the expense of time. The trade-off between stability and over-learning can be addressed by properly choosing the leniency of the criterion. For this paper, the criterion used was to cut optimization after the maximum value (where 'value' means the average fitness of the sample produced in a particular generation) had not increased for \( N \) generations. So if \( N = 4 \), and the last 5 values are 3, 2, 4, 1, 2, then if the next value is less than 4, the criterion will be satisfied and the optimization will stop. If the next value is at least 4, then the count will start over again and the optimization will continue for at least 3 additional rounds. In experiments, \( N = 4 \) was fairly modest, somewhat increasing stability without slowing the algorithm down too much, whereas \( N = 10 \) was fairly large, generally getting activated only when substantial progress had been made towards the optimal policy or when the policy was already at a plateau.

Two types of transformation schedule were developed to explore the trade-off. The first was called 'morph'. Because the hand tuned model is not robust enough to start transforming off the bat, the morph schedule optimizes until it reaches an initial light plateau, with \( N = 4 \). After that, it transforms every generation, at a constant rate set by the human. Once it reaches the target model, it will optimize until it hits an \( N = 6 \) plateau. This schedule is called 'morph' because it is fairly transformation heavy, making constant progress every generation and leaving most of the optimization till the end. This strategy completely avoids overlearning, but also bypasses any stability improvements.

The other strategy is called 'opt'. This is because it tries to optimize to a plateau between every transformation. There are two variants. The first is 'opt4'; it will optimize until it hits an \( N = 4 \) plateau, then transform by a fixed, predetermined amount (just as morph did). When it reaches the target it goes for an \( N = 10 \) before stopping. The second is 'opt10'; it goes for an
$N = 10$ plateau between each constant transformation. At the end, it optimizes til it hits an $N = 13$ plateau. The final optimization plateau in each case is not particularly important, since optimizing at that point is pretty much the same as optimizing the original Nao. What is important is whether it reaches the target with a viable policy distribution.

The three schedules were tested against the control. The control consisted of no schedule - it started with the target model and the hand tuned policy and attempted to optimize without any transformation. The opt4 strategy transformed at a rate of 4% of the difference each transformation, opt10 went 10% of the distance at a time, and morph went at rate of 2%. For each percentage of transformation, the upper leg is shortened by 0.3 mm and the lower leg is shortened by 0.45 mm. Since densities remain constant, the mass of each limb decreases correspondingly. The initial seed and sample size were the same as in the optimization of the original agent. Figure 11 shows the results.

**Fig. 11.** Transformation experiment results.

For the opt schedules, only generations that immediately follow a transformation, as well as the first and last generations, are plotted, so as to enhance readability. The control is hard to find, but it appears as a small pink blip in the bottom left of the chart. The control was unable to evaluate the sample produced by CMA-ES in the 10th generation. Evaluations were run on a large distributed computing cluster, with unreliable nodes and heavy traffic from other
users. Thus, the optimization program was designed to cancel and retry evaluating a sample if an evaluation run was going too slowly. It would also conclude an evaluation once more than 130 of the 150 samples had completed, assuming that the unfinished ones had poor fitness. It was known from prior test optimizations that some invalid policies would flail and contort the robot in such a manner as to interfere with the evaluator’s attempts to reset them for a subsequent run. The evaluator would ensure that the robot was standing upright before each run, so a robot that broke this functionality would never complete its evaluation. It was found that some policies consistently exhibited this behavior, so that if too many such policies occur in the same sample, the optimization will never be able to evaluate that sample and will be stuck. It was also found that all such policies were completely unable to get up, or to even perform any intermediary step. This situation had occurred in several trial optimizations, and a consistent observation was that the problem only occurred after the optimization had already diverged. In the particular case of the control for this experiment, of the 1500 different policy/model combinations evaluated during its short life, not a single one was capable of getting on its feet or even of scoring a fitness better than -4.4. None of the robots would be able to perform any task that would earn them a meaningful reward, so there would be no information for them to use in the pursuit of optimization. Thus, the optimization was clearly doomed to fail. This shows that the control is unable to accomplish the task.

The morph schedule also never made much progress. It ran into the same halting situation as the control, but ironically, only after it had transformed all the way to the target model. Even if the optimization had had better sampling luck in the last few generations, it may have failed to find a viable policy, seeing as its fitness was very low during the entire run. This algorithm reached the target model very quickly, but at the cost of failing the optimization task. Whether the failure was due to too fast of a transformation rate or because of the nature of the morph schedule is a matter for future work.

Both opt schedules succeeded at the task. Some policies from the final generations were checked manually to ensure a false solution had not developed. Opt4 completed in about half as many evaluations as opt10, but had lower final fitness. However, as previously discussed, it was given less time at the end to optimize, so its final fitness is not necessarily meaningful. This shows that a small amount of optimization between each transformation may strike a proper balance between speed and cautiousness. On the other hand, the fact that opt10 succeeded, whereas morph failed, suggests that in this domain overlearning is not a serious threat, but that optimizing for stability and robustness is a must. The fact that opt10 would perform more than 10 optimizations on a single model, and then successfully make a large jump to a fairly different model, suggests that the techniques used by the different models are similar and that improvements to one model can likely be exploited by others. It also seems apparent that policies which are highly stable and fast on one model will be more likely to successfully get up on another model. Otherwise, extensively optimizing for speed of get up would not help prevent incompetence when the model is changed.
to a very different one. Why fast get ups are more robust to changes in the model is intriguing and would be valuable future work.

7 Future Work

As suggested above, exploring why fast policies are more likely than slow policies to be viable in very different models would make for intriguing future research. Also, while the experiment in this paper seems to find that diverse models tend to have similar optimal policies, it could be the case that other target models show the opposite trend. It would be interesting to see if there are cases where overlearning is particularly dangerous, and what circumstances engender this scenario.

Another approach to future work would be to try to answer the question of whether certain models are incapable of getting up at all. As mentioned previously, there are some cases where any prospect of getting up is slim and would require a very novel technique. However, proving theoretically that a get up routine is impossible for some model, even if there are strong constraints put on the nature of the get up routine, would be very difficult. Also, some issues such as short arms are more obvious than others, and there may be some very subtle but important conditions under which a get up routine is impossible.

8 Acknowledgements

I would like to greatly thank Peter Stone and Patrick MacAlpine for the endless guidance and support they have provided me. I would also like to thank Adrian Lopez-Mobilia for first proposing the idea of transforming the model during policy optimization. Finally, I greatly appreciate all the love and support of Lucy Liang, as she has helped me stay focused, motivated, and cheerful.

References