QuadraTot

Final Report

Diana Hidalgo, B.S. CS '12; Sarah Nguyen, B.A. CS '13; Jason Yosinski, M.S. CS '11
{djh283,smn64,jy495}@cornell.edu

December 1, 2010

1 Abstract

This paper presents several approaches to optimizing a quadrupedal trot gait for forward speed. Given a parameterized walk designed for a quadruped robot designed and printed in the Cornell Computational Synthesis Lab, we implement, test, and compare different learning strategies, including uniform and Gaussian random hill climbing, a form of policy gradient reinforcement learning[**?**], Simplex (Nelder-Mead), linear regression, SVM regression, and an evolutionary neural network (HyperNEAT)[**?**]. Because the fastest learned walk was not significantly faster than the fastest randomly generated walk, we conjecture that the motion representation for the robot is more integral to forward speed than the learning algorithm.

Keywords: Learning Control, Walking Robots, Multi Legged Robots

2 Introduction

We are using machine learning to design a gait for a quadruped robot. The robot has an on-board computer, drivers for the motors, and will have camera feedback. The code so far consists of the robot class, optimization class, parameterized model, sine model, and camera feedback. Both the hardware and software are progressing on schedule. Some promising gaits have been generated, and we anticipate a final evaluation of the robots gait in November.

3 Problem definition

We are testing several different learning methods to design a gait for a quadraped robot from the Cornell Computational Synthesis Lab. The metric for evaluation of the designed gait is

speed. A comparison and evaluation of the many different methods available for optimizing the gait of legged robots will be useful for future work on this challenging multidimensional control problem.

4 Method

We implemented and tested 8 different learning strategies:

- *Uniform random hill climbing*: Creates a random neighbor by randomly choosing one parameter to adjust, and changing it completely randomly in UniformStrategy(). Then evaluates this neighbor by running the robot with the newly chosen parameters. If this neighbor results in a longer distance walked then the previous best neighbor, we save this neighbor as the new best neighbor. We then repeat the whole process using the best neighbor as a base.
- *Gaussian random hill climbing*: Creates a random neighbor by changing each parameter randomly based on a normal distribution. Then evaluates this neighbor similarly to uniform random hill climbing.
- *N-dimensional policy gradient descent*: Estimates the policy gradient by evaluating t randomly generated policies near an initial parameter vector. Then computes the average score for each parameter and adjusts the base policy according to the estimated gradient.
- *Random*: Randomly generates policies in the range of the motion representation.
- *Simplex (Nelder-Mead) Method*
- *Linear regression*: To initialize, chooses five random points and trains on them using least-squares. Then in a loop, takes a fixed-size step in the direction of the gradient, and retrains on all points so far.
- *SVM regression*
- *Evolutionary Neural Network (HyperNEAT)[**?**]*

5 Related work

6 System Architecture and Implementation

The quadraped robot has an on-board computer running Linux. The lower level drivers are in C and we are implementing the system in Python. Feedback about distance travelled is provided via a Wii remote.

- `Robot` class: Class wrapper for commanding motion of the robot. The `Robot` class takes care of the robot initialization, communication with the servos, and timing of the runs. In addition, it prevents the servos from ever being commanded to a point outside their normal range (0 - 1023) as well as beyond points where limbs would collide with parts of the robot body. The main class function, `run`, accepts a motion model (any function that takes a time argument and outputs a 9 dimensional position) and will run the robot using this motion model, including, if desired, smooth interpolation over time for the beginning and end of the run.

- `Strategy` and `RunManager` class: The user has a choice between seven different learning strategies: uniform random hill climbing, Gaussian random hill climbing, N-dimensional policy gradient descent, random, simplex, linear regression/prediction, and SVM regression/prediction.

- `Parameterized Motion Model` abstract base class: Several functions exist, but none are within a class framework.

- `SineModel` classes: Commands motors to positions based on a sine wave, creating a periodic pattern. Parameters are: amplitude, wavelength, scale inner vs outer motors, scale left vs right motors, scale back vs front motors. Currently a function, not a class.

- Other derived motion model classes: Several versions of a SineModel exist, each with different parameters, though solely as functions.

- `WiiTrackClient` and `WiiTrackServer` classes and hardware: A Wii remote tracks the location of the robot through an infrared LED mounted on top of the robot. We used CWiid library which is able to interface with the remote via bluetooth to determine the location of the infrared LED. This information is accessed through some functions we wrote and passed to the program. The program gets the robot's position at the beginning of each run and then again at the end. The program then calculates the net change in position and uses that as the distance walked.

7 Experimental (or Theoretical) Evaluation

7.1 Methodology

- Each algorithm was run on 3 different initial parameter vectors on the physical robot. We stop each run after plateauing results (no improvement for half the policies seen so far). We evaluate each method based on the amount of improvement seen from the initial parameter vectors, and on the fastest speeds achieved during runs.

7.2 Results

- We have done runs of 3 different inital parameter vectors for random hill climbing, gaussian random hill climbing, policy gradient descent, random, simplex, and linear regression. We developed several gaits that were about 4 times faster than the original hand-coded gait.
- We also coded `explore_dimensions` to collect data for dimension varying plots. We intend to do further explore this by running `explore_dimension` around a parameter that produces a medium-good gait.

7.3 Discussion

8 Future work

- Different motion representations
- Evolutionary algorithms/HyperNEAT[**?**]
- Learning how to turn
- Open sourcing code for others to extend

9 Conclusion

Because the fastest learned walk was not significantly faster than the fastest randomly generated walk, we conjecture that the motion representation for the robot is more integral to forward speed than the learning algorithm.

10 References

11 Acknowledgments

- Hod Lipson, Cornell Computational Synthesis Lab: advisor
- Jim Tørreson, University of Oslo: advisor
- Juan Zagal, University of Chile: designed and printed robot and provided assistance
- Jeff Clune, CCSL Red Couch: collaborated on HyperNEAT implementation/testing
- Cooper Bills, Cornell University: assisted with Wii tracker development
- Anshumali Srivastava, Cornell University: Teaching Assistant

12 The Team

- Diana Hidalgo:
  - Simplex (Nelder-Mead)
  - `WiiTrackClient` and `WiiTrackServer` classes and hardware
  - Plotting and evaluating results
- Sarah Nguyen
  - Random
  - Uniform random hill climbing
  - Gaussian random hill climbing
  - N-dimensional policy gradient descent
  - Linear regression and prediction
  - `RunManager` and `Strategy` classes
  - Video editing
- Jason Yosinski
  - SVM regression and prediction
  - Evolutionary Neural Network (HyperNEAT)
  - `WiiTrackClient` and `WiiTrackServer` classes and hardware
  - `Robot`, `RunManager`, and `Strategy` classes
  - `Parameterized Motion Model` abstract base class
  - `SineModel` classes
  - `explore_dimensions`
  - Plotting and evaluating results

13 Appendix

A brief description of the code uploaded to the CMS follows:

- `optimize.py`: Determines a strategy to try and runs the robot with that strategy.
- `RunManager.py`: Deals with all the details of running the robot, including choosing an initial parameter, running the robot multiple times, tracking distance walked, and writing to the log file. Also includes the `explore_dimensions` method.

- `Strategy.py`: Contains all the different possible strategies, which will be passed as objects in `optimize.py`.
- `Robot.py`: Implements the `Robot` class, described in Section **??**.
- `SineModel.py`: Implements a sine based motion model, described in Section **??**.
- `Motion.py`: Motion helper functions.
- `WiiTrackServer.py`: Broadcasts the position of the infrared LED¿
- `WiiTrackClient.py`: Connects to the WiiTrackServer to get the current position information.