Training a Robot to do Cartwheels

Nicholas Bone
Department of Computer Science, Western Washington University, USA

Abstract

This paper introduces "Daddy Long Legs", a robot designed to be capable of learning to propel itself via cartwheels. The robot is built using components from the LEGO® Mindstorms® NXT kit, and programmed using the RobotC language and development environment. It has three long legs radiating symmetrically from a compact central body, with a touch sensor at the end of each leg so it can sense which of its "feet" are on the ground. It learns a state-value function using delayed reinforcement from the touch sensors to estimate the relative values of different limb positions. At each time step it considers several possible actions (motor powers to each leg) and chooses whichever action leads to the best estimated value for the predicted resultant state. It also supports a supervised training mode in which it lets a human move its limbs to help direct early learning. Experimental results show definite learning progress, but failure to achieve fully-autonomous cartwheels within the time constraints of the experimental setup. The strengths and weaknesses of the physical design and the learning algorithms are analyzed, and improvements are suggested for future work.

1 Introduction

For this project I set out to build and train a robot to do cartwheels. Cartwheels are a complex behavior requiring balance, momentum, and limb coordination. To perform a cartwheel, one must fall to the side while simultaneously generating enough rotational momentum to continue the spin so that each limb is in position to support the body in turn and prevent it from collapsing. This is easiest to achieve when there is relatively high concentration of mass near the extremities of the limbs, or when the legs are long and the body is small and light. Humans learn cartwheels through coaching and proprioceptive feedback—can a robot do the same?

My goals for this project were as follows:

- Build a multi-legged robot that is physically capable of locomotion via cartwheels—that is, rotating its body unidirectionally in the vertical plane to achieve the affect of a sequence of "steps" with its legs, while keeping its body off the ground.
- Employ reinforcement learning to train the target behavior.

This implies a learning task in which the robot initially has no knowledge of how to do cartwheels, or even to walk at all, but through experience gradually becomes more adept at propelling itself forward in a cartwheel motion, ideally without falling over. The training experience may be either supervised or autonomous, or a mix of the two. This is analogous to how a human learns: first by being shown, then with verbal or physical guidance from a coach to achieve successive approximations of the target behavior, and finally through proprioceptive feedback and outcome analysis to reinforce successful actions.

I attempted to solve this problem using a minimalistic three-legged robot guided only by proprioceptive feedback from its motors and reinforcement signals from feeling which of its feet are touching the ground. In order to limit the search space, the robot learns a state-value function to aid in action selection, rather than estimating action values or directly searching the policy space. In order to mimic human learning, I implemented a supervised training mode in addition to the autonomous learning mode, allowing a human "coach" to physically manipulate the robot towards the target behavior.

I chose this project primarily because it seemed like a good candidate for reinforcement learning, since the "cartwheel" behavior is difficult to manually write a program for. My other main motivation was simply curiosity as to whether I could get it to work using the limited tools available.

Humans continually learn by reinforcement, but applying such learning in robotics remains an elusive prospect, requiring much design and experimentation with each new application. Learning effective locomotion using reinforcement would be a valuable skill in many robotics applications. The study of reinforcement learning in robots also helps to deepen our understanding of how humans and other animals learn.

2 Related Research

I came up with the idea of training a robot to do cartwheels while brainstorming fun and challenging applications that could make good use of reinforcement learning and be feasible with the tools I had available. I built the robot and devised the control algorithm without reference to any previous work. However, I was inspired in my brainstorming by the works of Daniele Benedettelli, in particular by his "Resonant Biped" [1], which oscillates a large weight (the NXT brick) from side to side to help achieve a walking gait. Thinking about this led me to wonder if the oscillation could be made large enough to initiate a cartwheel.

After completing this project I searched online but found no other attempts to build minimal (and inexpensive) learning robots capable of doing cartwheels. There are several videos on YouTube of robots performing cartwheels (e.g. [2], [3], [4]), but those all feature multi-jointed limbs and relatively expensive hardware, and the similarity of the behaviors across videos suggests that it was hardcoded by the manufacturer and involves no learning on the part of the robot.

There have been many papers about building learning robots, but apparently none about learning to do cartwheels. Perhaps the most applicable prior research, because of its focus on reinforcement learning to control balance, is the research that has been done on the inverted pendulum—or "pole balancing"—task, such as [5]. In this work, Hougen et al train a minimal robot to keep an attached pole from falling over using online reinforcement learning and a neural network representation. The specific learning algorithms they used differ from mine, but their minimalistic approach both in the hardware and in the input/output representation is similar to mine.

The learning algorithms I used in this project are well-established in the machine learning literature, but they are mostly used in software-only applications, and I found no published uses of them to teach a simple legged robot to perform complex locomotion such as cartwheels. The learning algorithms I used are all pretty standard, and may be found in Mitchell [6].

3 Tools

I used exclusively LEGO® Mindstorms® NXT components to build my robot, as this was the only hardware I had available.

For software there was a somewhat greater choice, both in programming language and development environment. I first tried "Not eXactly C" (NXC) using the Bricx Command Center (BricxCC) IDE [7], but found NXC to be severely limiting in its missing support for floating point arithmetic and user-defined structures, and the BricxCC editor is buggy and user-unfriendly. After some online research ([8] was particularly helpful), I decided to try RobotC [9], which supports both floating point arithmetic and user-defined structures, and has a built-in debugger (although it requires custom firmware on the NXT). Unfortunately, RobotC is also quite buggy, and has very limited and often wrong documentation, but all-in-all it's better-suited to a complicated project than NXC/BrixcCC, and the integrated debugger has been useful for monitoring network weights during training without having to write them to the NXT display.

4 Methods

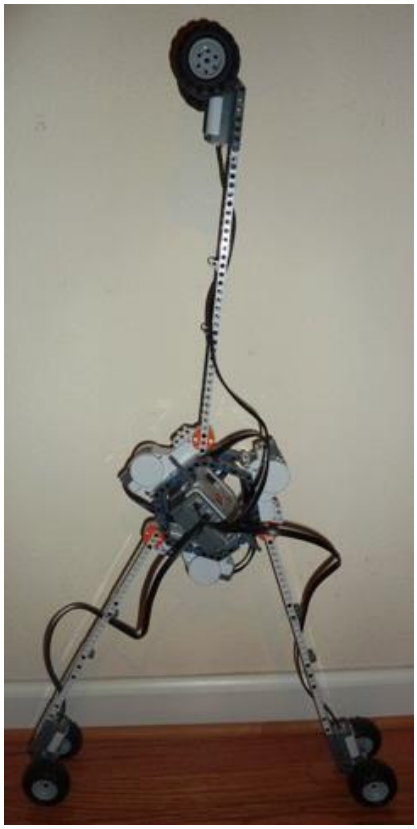
In this section I describe the design of my robot and the algorithms I used to program it. It is divided into four sub-sections:

- "**Robot Design**" describes the physical construction of the robot, from a high level.
- "**I/O Representation**" explains how the robot translates its raw sensor readings into an internal state representation for learning.
- "**Training**" describes the learning algorithms I use, again from a high level.
- "**Program Control Flow**" specifies how the robot behaves, and its different operational modes.

4.1 Robot Design

Cartwheels are physically difficult even for humans, and take a lot of practice to do well. Building a robot capable of cartwheels is a challenge, especially given the constraints imposed by the NXT components: the brick and motors are relatively heavy, whereas the plastic LEGO® pieces are light and relatively weak when chained together to make a long limb. Achieving and maintaining an appropriate momentum is also difficult for the robot, especially with no gyroscopic sensor. Therefore I attempted to build a robot capable of "slow cartwheels": start in a balanced position, then initiate a fall by shifting weight to one side, catch the fall with a free limb, and return to a balanced state using a different set of limbs. Given the relative weights of the LEGO® NXT components, this required making very long legs, which unfortunately made the robot laterally unstable.

For simplicity and to keep the size and weight down I decided to use a single NXT brick, which has four sensor ports and three motor ports. With only three motors there can be only three legs—since each one must be independently operable—with each leg powered by a single hip motor. Three is also the minimum number required to do cartwheels, as—when in a cartwheel—the "top" leg must be used to shift weight from the "back" leg to the "front" leg. I also wanted to keep the design as symmetrical as possible, so that each leg is effectively interchangeable.



In order to learn, the robot must have sensors so it can get a reward signal from the environment. It can get proprioceptive feedback from the motors to discern the rotations of each hip joint, but it also needs to know its orientation relative to the ground plane. Therefore I put a touch sensor ("toe") at the end of each leg so that the robot can sense which of its "feet" are on the ground.

My complete robot is shown at right. The NXT brick is mounted in a central hexagonal chassis, surrounded by the three motors. Each leg has approximately 180 degrees of rotational freedom. The wheels at each foot are on fixed axles, and are there for the softness and grip the rubber tires provide.

4.2 I/O Representation

The robot has six sensors: three touch sensors, and three motor rotation positions. Due to the symmetry of the robot, we factor out the touch sensors from the input representation so the robot has just two operational states: one toe touching or two toes touching (cases where zero or three toes are touching are considered "non operational"). The hip rotations are translated based on which toes are touching so that the rearmost touching leg is always "leg 1" in the input, the foremost touching leg (or the next leg we want to touch given the direction of the cartwheel) is "leg 2", and the going-over-the-top leg is "leg 3". This allows both for a simpler input representation and for faster learning, provided that the balance of the robot is close enough to symmetric.

The three motor rotation positions are normalized into floating point values between 0 and 1, where 0 represents the minimum leg angle and 1 represents the maximum (these range bounds are determined via manual calibration when the program starts, as described in the "**Program Control Flow**" section below). Each of these leg angle values is then translated into five inputs via a Gaussian transformation, $e^{-B|x-c|}$, with evenly spaced centers, $c = \{0.0, 0.25, 0.5, 0.75, 1.0\}$, and $B = 4$. This ensures that the inputs to the learning algorithm are symmetric so that one end of the range is not artificially favored over the other, and that each input represents a single concept, in this case: how close the leg is to that degree of rotation.

Thus, there are effectively 16 inputs in the internal state representation: 15 to encode the leg angles, and 1 to indicate whether one toe is touching or two. For simplicity, I treat the two operational states as separate learning tasks, each with 15 inputs. The robot's outputs are simply the power levels to send to each motor, in the range [-100, 100].

4.3 Training

Reinforcement Learning is central to my project, so the robot needs to be able to sense when it is performing well. Each time it lifts the rear toe off the ground (when standing on two legs) or touches the next toe down (when standing on one leg), it receives a maximal reward. Conversely, if the front toe lifts or the rear toe touches again, it gets a minimal reward (punishment). This of course implies that there is a single prescribed "correct" direction for the cartwheel—in the image of the robot above, the correct direction is to the right.

Just like with humans, the robot cannot be expected to stumble upon the target behavior in any reasonable amount of time without some guidance from a coach, so I programmed a special "training" mode in which the robot goes limp and lets the human handler move its legs. In this mode, the robot gets a constant moderate reward (assuming that the "coach" will always put it in a desirable state) in addition to the normal toe-touch rewards.

As noted in the previous section, the two operational states are treated as separate learning tasks. For each task, the robot learns a state-value function mapping the 15 inputs to a single relative value estimate for that state. When a reward is received, the state-value estimate for the preceding state is adjusted in the direction of the reward value. In the absence of a reward signal, the robot continually updates its estimate of the current state value in the direction of the value estimate for the next state it

encounters, decayed very slightly (multiplied by 0.99) under the assumption that the next state ought to be better than the current state.

Initially I approximated the state-value functions using 3-layer fully-connected feed-forward artificial neural networks with three hidden units and sigmoid activation at the hidden layer and output layer. However, as I got into debugging, I decided that I should start more simply, especially since I had not yet proven that I needed the representational power provided by hidden nodes. So my final version uses only 2-layer perceptron-style networks with linear activation at the output. This representation is simpler and faster to train, but assumes that the individual leg angles are independent in their effects on the value of the state, which probably isn't quite true. The network weights are updated according to standard backpropagation procedures (which, in the 2-layer case, just uses the LMS update rule) using a learning rate of 0.02.

4.4 Program Control Flow

When the robot is first turned on, the legs must be calibrated, so that the input values may be normalized accurately. This is done by manually tapping each toe at either extreme of leg angle. I chose to have this as a manual step because the legs must be physically moved in order to do this accurately, and this gives the human the option to restrict the movement range of each leg if desired.

After calibration the robot is in "neutral", and there's a main menu with four options: autonomous mode; training mode; self-training mode; or quit. Each of the three operational modes, when selected, will run until the exit button is pressed, which will return the robot to "neutral" and redisplay the main menu. Given my definition of "operational state", the robot will neither act nor learn whenever fewer than one or more than two toes are touching (however it will resume acting and/or learning the moment the toe-state returns to "operational").

The basic control flow of the robot, when acting in autonomous mode, is as follows:

1. Robot senses physical state and translates to internal representation.
2. Robot considers possible next states (different angles of its legs) and generates a value estimate for each by activating the appropriate ANN, choosing the next state with the greatest activation.
3. Robot sets motor powers in order to move towards chosen next state, then waits for $1/20^{th}$ of a second to give the motors a chance to take effect.
4. Robot senses physical state and collects training signal. Either:
    a. Gets a reward (or punishment) if the touch sensors changed; or
    b. Activates the ANN to get the value estimate for the current state, which is then decayed.
5. Robot trains ANN on previous state (step 1) using collected training signal.

In "training" mode, it's simpler:

1. Robot senses physical state and translates to internal representation.
2. Robot waits for $1/20^{th}$ of a second while human moves it.
3. Robot senses physical state and collects training signal. Either:
    a. Gets a reward (or punishment) if the touch sensors changed; or
    b. Gets constant moderate "training" reward.
4. Robot trains ANN on previous state (step 1) using collected training signal.

The "self-training" program follows a simple hardcoded strategy of fixed target leg angles given the current toe configuration. This is like the human training mode in that the learner isn't in control of the motors (the trainer program is), but it uses the same reward structure as in autonomous operation (i.e. the special "training" reward is turned off).

Note that autonomous mode is the slowest in terms of updates per second, because it has to activate the ANN many times in order to determine the "best" next state. In practice, this was on the order of twice as slow as manual training mode.

5 Experimental Results

Given the nature of my robot it was difficult to perform controlled experiments. I did not implement any long-term memory (saving/loading a trained network to/from the file system), and the network weights were randomly initialized at program startup, so each test was different. Furthermore, given the instability of the robot, I always had to manually support it (more or less), and it never achieved the ability to perform a truly autonomous cartwheel. Therefore, performance metrics like "number of successive cartwheels without falling over" were out of the question.

However, over many repeated trials I subjectively observed improvement in the robot's behavior after exposure to training. This was my basic informal test procedure:

1. Turn on robot and calibrate legs.
2. Activate autonomous mode and observe behavior (while holding robot so it doesn't fall).
    a. Expected behavior: random leg movements.
    b. Observed behavior: same.
3. Activate training mode and guide robot through a sequence of cartwheels, anywhere from 5 to 20. (I did not, as a rule, let the robot learn purely through experimentation, because in most cases the initial behavior would have to be forcibly overridden in order to get it to bend in such a way as to receive appropriate reinforcement signals.)
4. Re-activate autonomous mode and observe behavior.
    a. Expected behavior: robot favors the ultimate positions of each operational state (that is, whatever position immediately preceded the reward and state-change associated with the lifting of the rear toe or the setting of the next toe).
    b. Observed behavior: almost always a noticeable improvement over the initial behavior in the direction of the expected behavior, appearing to perform the expected behavior more than half of the time, and coming close to it most of the rest of the time.
5. Repeat the training/testing steps as desired, optionally substituting "self-training" mode for the manual training mode.

Based on my observations, I draw the following conclusions:

› The robot definitely learns based on training, and the learned behaviors, while not being completely effective, appear to be significantly closer to the target behavior than do the initial random behaviors.

- The simpler 2-layer network (no hidden units) learns faster and more accurately—at least in early training—than my initial 3-layer network with three fully-connected hidden units. By monitoring the current state-value estimates (which I output to the NXT display), it appeared that the 3-layer network over-generalized the training signal and failed to appropriately discriminate between disparate states. I suspect that this was at least partly due to the "catastrophic forgetting" problem endemic to sequential neural network training [10].
- Manual training mode seems to be most effective when the robot is quickly moved to the target limb configuration for the current operational state (number of touching toes), and then held in that position for a moment before transitioning to the next state and collecting the touch-triggered reward. I believe this is because of the constant training reward in this mode, and the moderately slow learning rate. The longer the robot is held in a position, the closer its state-value estimate for that position will approach the constant training reward, which is significantly higher than the maximum possible initial value of any state based on the random initialization.
- Given the previous point, and the fact that the "self-training" mode does not use the constant reward, manual training seems to be more effective (when done well) than self training. If the self-training program were capable of actually leading the robot through successive cartwheels without falling over then I presume it could be a more effective long-term trainer.
- I previously stated that because the robot has no conception of momentum, I wanted to build it so it was capable of "slow cartwheels". However, the way I coded the action selection, the robot typically moves its limbs fairly quickly, resulting in a jerky motion. Based on the physical design, I know that it's capable of initiating a fall in the target direction with slow movements only, however once the fall starts it needs to move rather fast in order to catch itself (either that or have really good balance on one leg). In my tests I observed that—even when it was moving its limbs in mostly the right direction for a cartwheel—if I were to stop supporting the body it would do the splits and fall flat. I suspect that, if the robot could be kept from falling sideways (due to its long shaky legs), the fast jerky cartwheel would be physically possible, but long-term learning would probably be more successful if I constrained the action selection so the limb motions were smoother and gentler.
- Given the length of the robot's legs (which doubled from my initial build), it would almost certainly be more effective to build wider feet for greater stability and weight.
- My tests were limited to a few minutes each, which is very little training time considering the complexity of the task. (Usually, after several minutes in autonomous mode the robot would turn itself off. I don't know if this was due to overexertion or perhaps an omission in my code where it was somehow failing to "keep alive".) However, given the instability of the robot, and the jerkiness of its movements, I suspect that my current program as-written is incapable of achieving fully-autonomous cartwheels. Longer-term training and testing would be necessary to answer this for sure.

6 Conclusion

For this project I built a robot that is almost capable of performing cartwheels. As discussed in the previous section, it's doubtful whether the final incarnation of the robot is actually physically and "mentally" capable of achieving fully-autonomous cartwheels, but it's demonstrably close. The greatest achievement of this project is in demonstrating the effectiveness of online reinforcement learning in a noisy "real-world" environment with a physical learning agent. It is quite remarkable, and surprising to me, how quickly the robot made noticeable improvements in its behavior only very brief training.

As noted above in the "**Related Research**" section, I found no previous work truly comparable to this project. However, my approach was notably different than most related research in that I employed dual-mode learning, with both supervised training and autonomous reinforcement learning. I believe that this dual-mode approach was instrumental in achieving the speed of learning I observed, and could be beneficial in a wide variety of reinforcement learning applications.

My robot showed good progress, but was far from being completely successful. Future work could improve upon this in many ways:

- Physically, the robot has several weaknesses to be addressed:
  - It is unstable; wider feet and reinforced legs could help with this.
  - The weight of the body is a burden; this cannot easily be fixed with NXT components.
  - The motors are slightly underpowered given the torques necessary in some positions, which sometimes delays or prevents the desired actions from being carried out. Again, this is a limitation of the NXT components.
  - Cartwheels would probably be physically easier with a greater number of limbs, or jointed limbs, however three rigid limbs as I used in this project ought to be enough.
- Currently, long-term training and testing is infeasible because the robot cannot save its state between runs. This would be a fairly easy feature to add to the program, and would facilitate more detailed and realistic experiments (a mere minute of training is too brief).
- The dual-mode learning seems to work well, but I did not perform experiments comparing it to learning using only one of the modes, so I have no results to prove its efficacy.
- I had planed (if I had more time) to add another level of learning, again inspired by the way humans learn. Currently, the robot just learns a state-value function, which is analogous to a human remembering which way to bend in order to perform a cartwheel. However, this requires a great deal of thought in order to select the next action. It should be fairly straightforward to add another neural network (or similar function approximator) to learn an action-value function, trained by the state-value function. When confidence becomes high enough (the actions suggested by the action-value function agree closely enough with those implied by the state-value estimation), the robot could switch to direct action selection, perhaps probabilistically based on the degree of confidence. This is analogous to human learning when a task becomes "rote" or "automatic"—after enough practice we can perform a cartwheel without thinking about it. This could be especially useful in more complex robotics applications

where the time it takes to select a good action based only on the state-value estimate is a performance handicap.

This project has given me a deeper appreciation of the intricacies and uncertainties of developing a real-world robot, yet has demonstrated that online learning can be remarkably effective even with those "real-world" handicaps. Further, it has shown that effective learning does not require state-of-the-art equipment, suggesting that truly useful, inexpensive, adaptive agents are within the grasp of current technology.

References

[1] Benedettelli, Daniele. "Resonant Biped" (web page). 2010. URL: http://robotics.benedettelli.com/ResonantBiped.htm. Retrieved 16 May 2010.

[2] (unknown). "robot cartwheels" (online video). URL: http://www.youtube.com/watch?v=nWW6xfyYnPw. Retrieved 8 June 2010.

[3] (unknown). "Bioloid cartwheel" (online video). URL: http://www.youtube.com/watch?v=B-OhwUHise8. Retrieved 8 June 2010.

[4] (unknown). "Meccano Robot Cartwheel.wmv" (online video). URL: http://www.youtube.com/watch?v=MVAelibpyGs. Retrieved 8 June 2010.

[5] Hougen, D., Fischer, J., Johnam, D. "A Neural Network Pole Balancer that Learns and Operates on a Real Robot in Real Time". *Proceedings of the MLC-COLT Workshop on Robot Learning*. 19 : 73-80. 1994.

[6] Mitchell, Tom M. *Machine Learning*. WCB McGraw-Hill, Boston. 1997.

[7] (various contributors). "Next Byte Codes and Not eXactly C" (web page and software). URL: http://bricxcc.sourceforge.net/nbc/. Retrieved 4 April 2010.

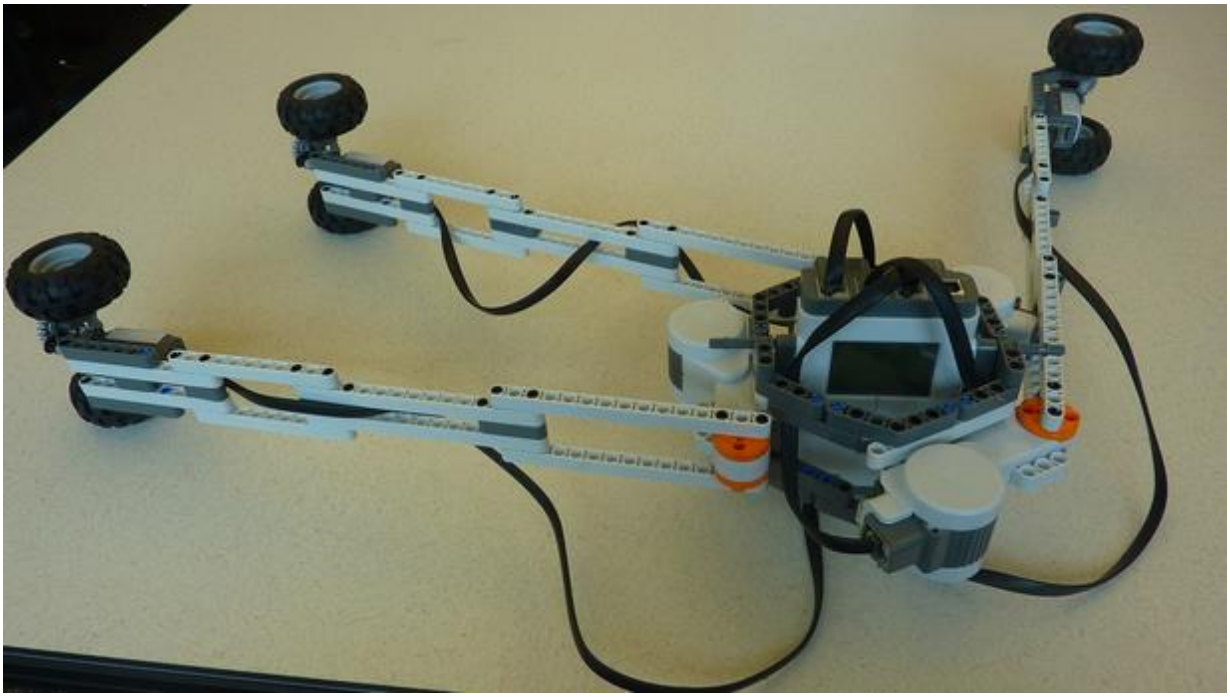
[8] Hassenplug, Steve. "NXT Programming Software" (web page). 2008. URL: http://www.teamhassenplug.org/NXT/NXTSoftware.html. Retrieved 18 May 2010.

[9] Robotics Academy. RobotC (software). 2009. URL: http://www.robotc.net/. Installed 23 May 2010.

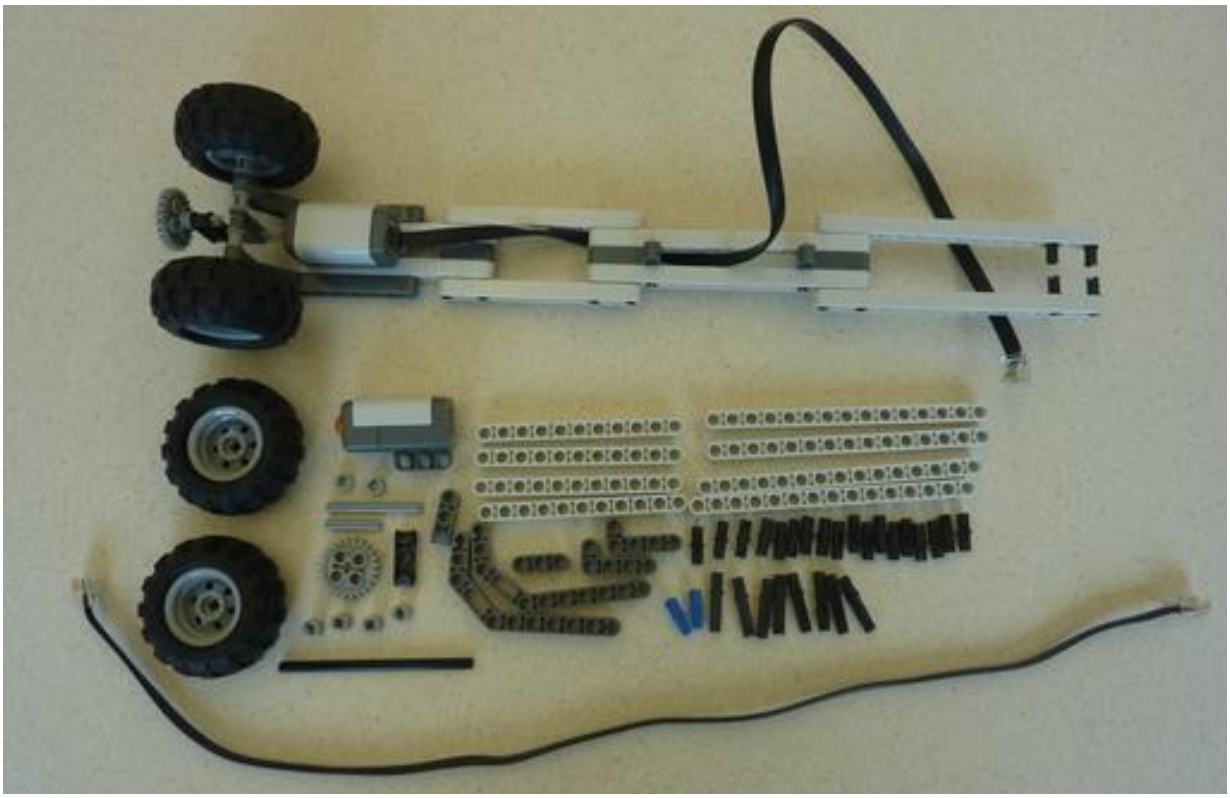
[10] Frean, M. & Robins, A. "Catastrophic forgetting in simple networks: An analysis of the pseudorehearsal solution". *Network: Computation in Neural Systems*, 10 : 227-236. 1999.

Appendix: Robot Building Instructions

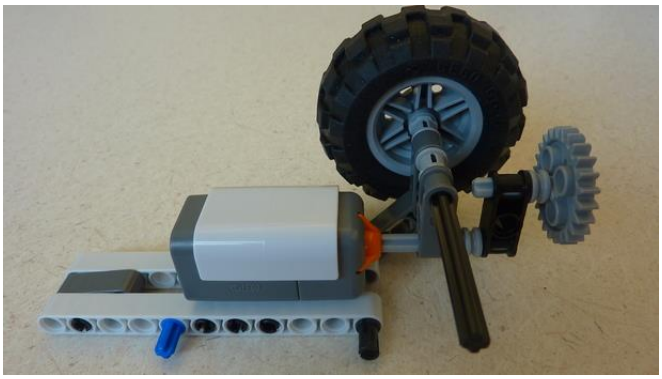
This appendix gives a pictorial guide to building the “Daddy Long Legs” robot.



The completed robot is shown above. Note that all legs are oriented so as to be rotationally symmetric.

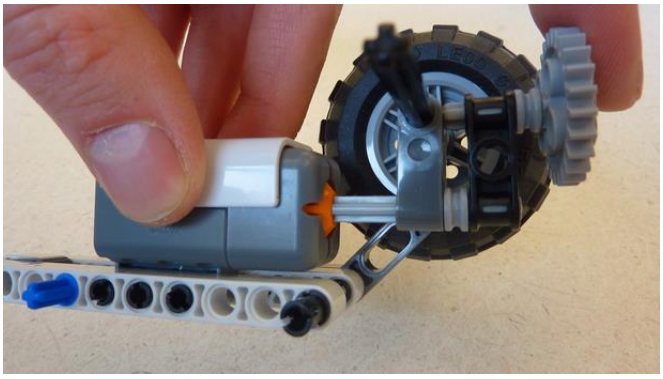


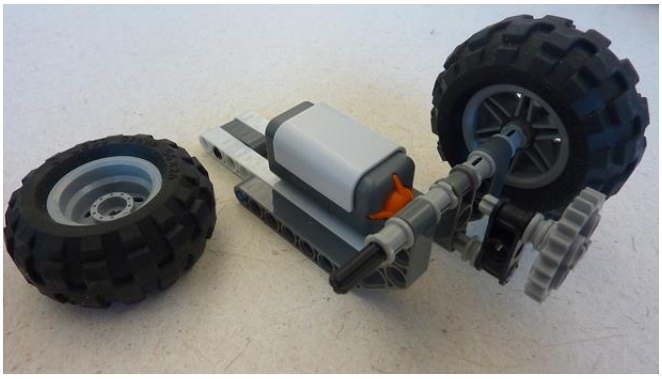
The previous picture shows a completed leg unit, together with all the pieces required to build it.



First build the foot and toe assembly, as shown in cross-section at left.

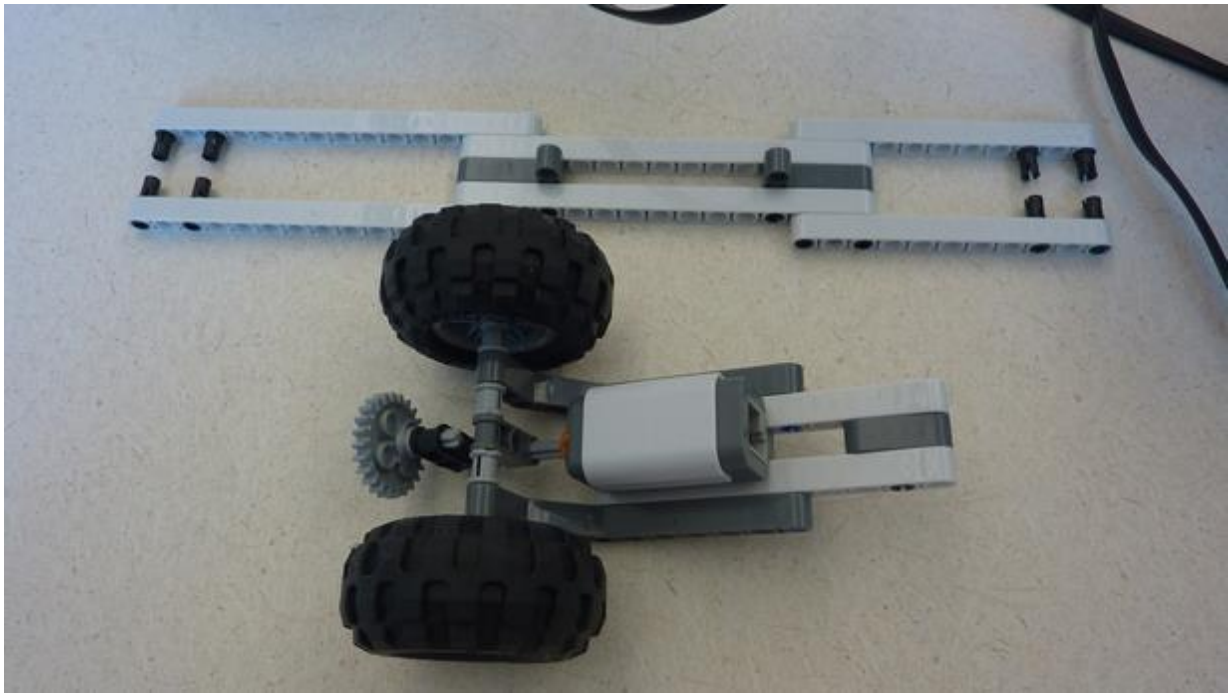
The touch sensor must be fully-depressible, as shown below.



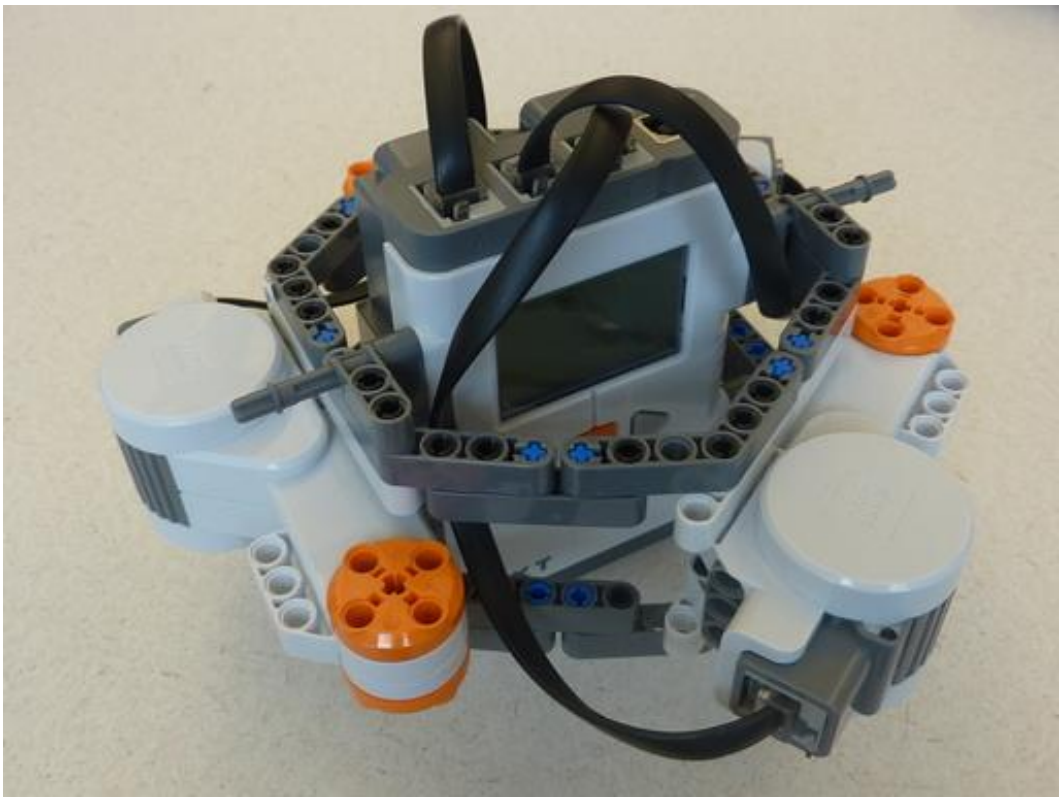


The nearly complete foot is shown at left.

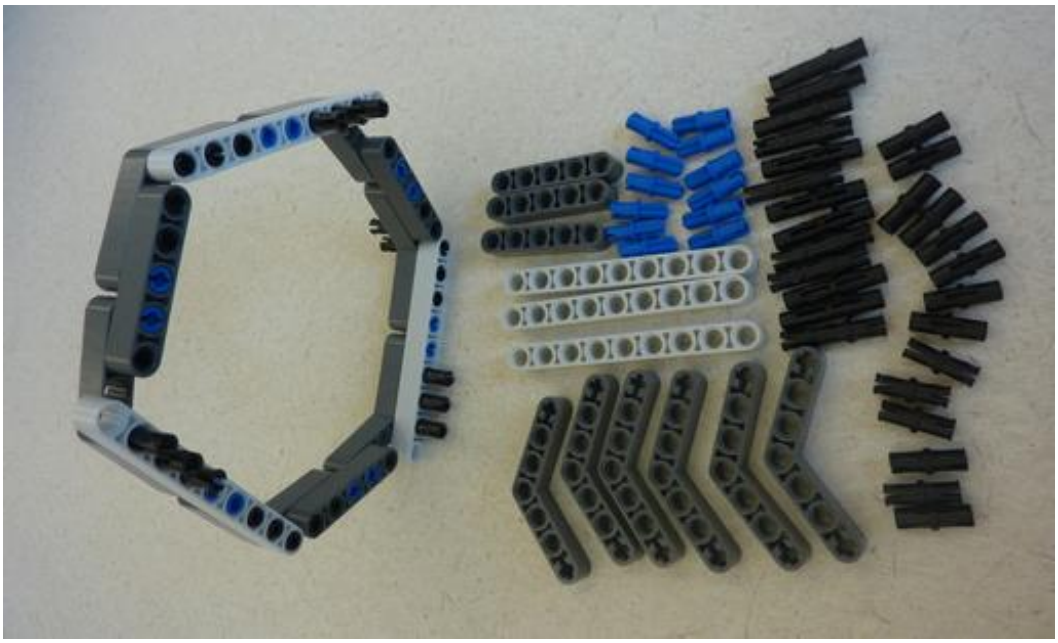
The rest of the leg is straightforward.

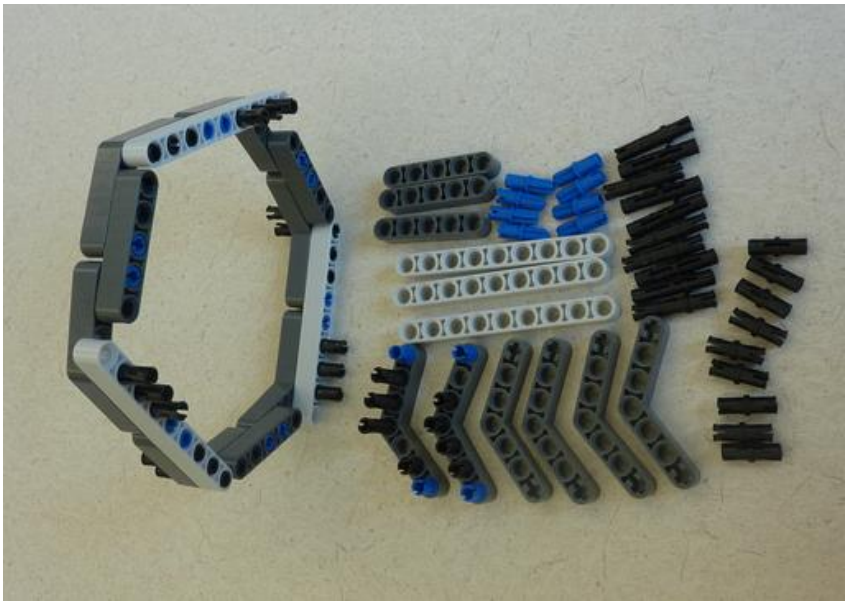


Once all three legs are complete, construct the body, shown completed below.

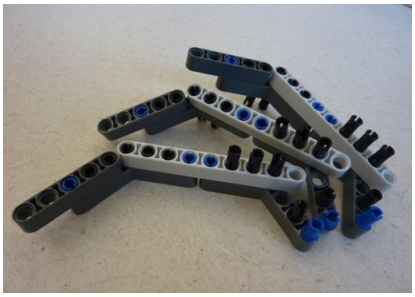


The skeleton of the body is comprised of two hexagonal assemblies; necessary parts shown below.

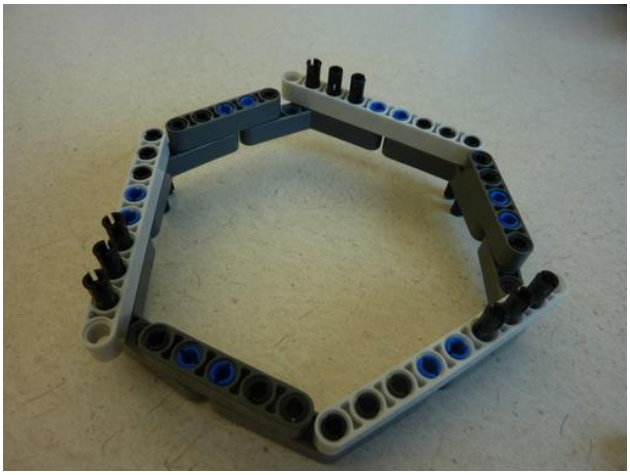


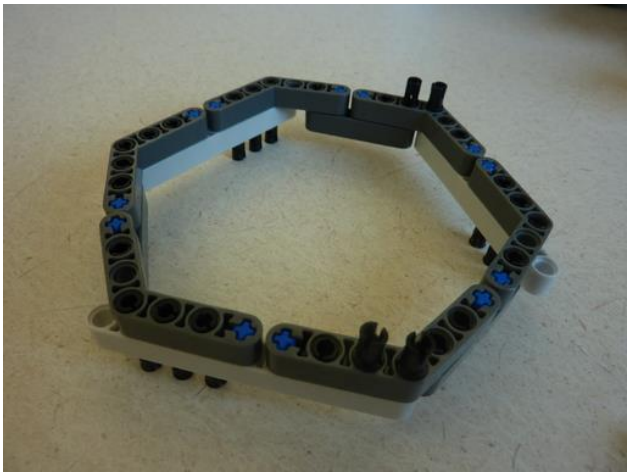


The image at left indicates how to begin assembling the hexagon, and below are the three symmetrical components of one hexagon. NOTE: the angles of the pieces are not perfect for hexagonal construction, so they have to be forced slightly.

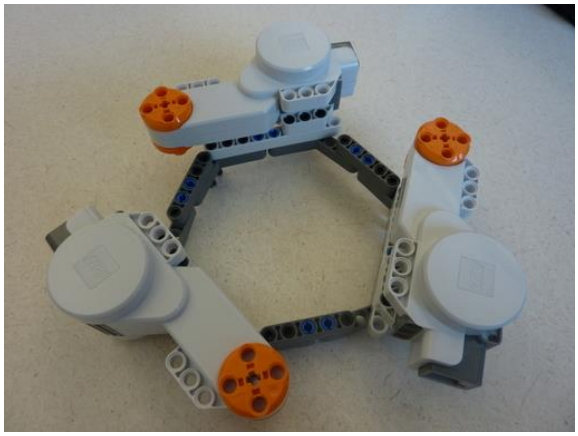


The two images below show the inside (left) and outside (right) views of a complete hexagon. The protruding pieces on the inside are for connecting to the three motors; on the outside, they are for connecting the the brick holder pieces. NOTE: the two hexagonal assemblies must be mirror images of each other—do not construct them to be identical or they will not fit!



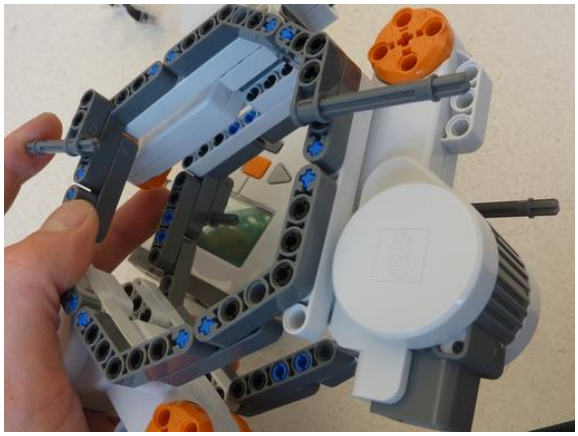


The image at right shows how to mount the motors onto one of the hexagonal assemblies. The other hexagonal assembly may then be attached to the top, sandwich style

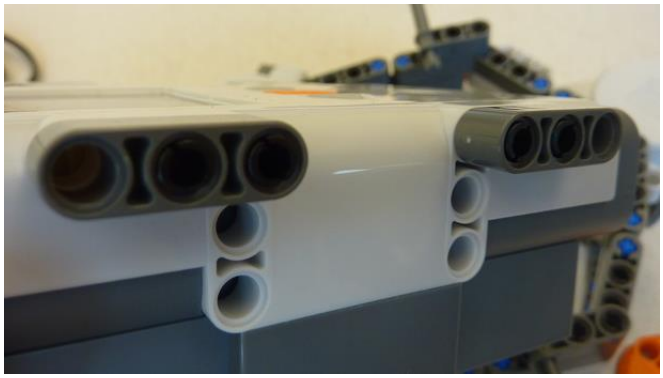


The image at left shows the NXT brick, and the pieces necessary to mount it to the central chassis.

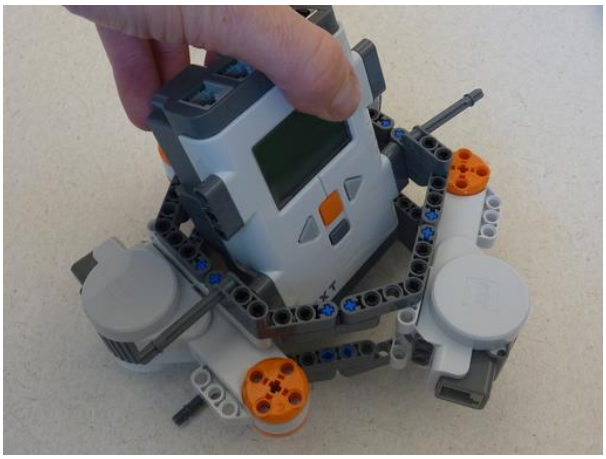
Attach the mounting pieces to the fully-assembled chassis as shown at right. The pins are designed to slide in and out so that the brick is easily removable (to change the batteries and such).

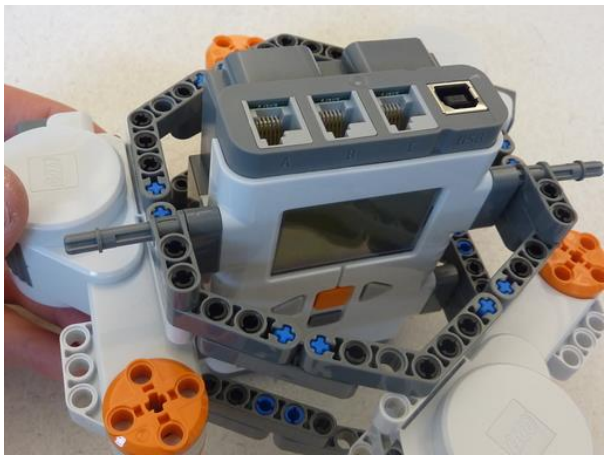


Attach the remainder of the mounting pieces directly to the brick as shown at right.

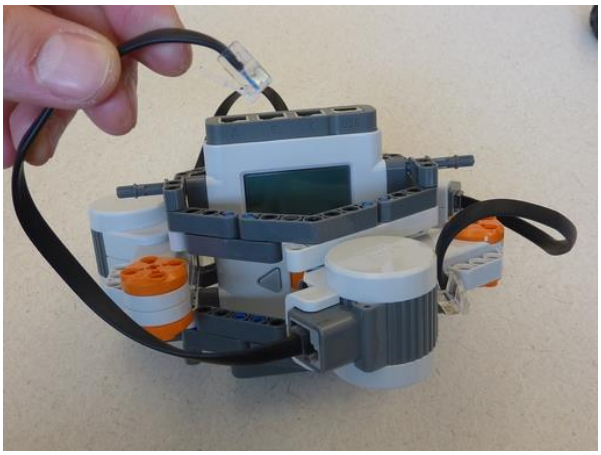


The brick may then be slid into the chassis and secured with the pins, as shown in the two images below.





Finally, attach the necessary cables from the motors to the brick, attach the legs, and connect the touch sensor cables.



NOTE: Ensure that the touch sensor from the leg connected to motor port A is connected to sensor port 1, B is connected to 2, and C to 3. This is a requirement for correct operation of the program.