

Q-Learning Hexapod (May 2009)

Matt R. Bunting, *Member, IEEE*, and John Rogers

Abstract—Reinforcement Learning techniques prove to be very interesting subjects in both control theory and cognitive sciences. In terms of control theory, building a system perfectly most likely becomes quite difficult, especially when considering sensorimotor errors. By building a system that learns how to accomplish a task on its own, there becomes no need to calculate and predict complex control algorithms. In cognitive sciences, the ability to learn is a core component of cognition. One such simple learning algorithm is the Q-learning algorithm. A six-legged (hexapod) robot will be implemented with a Q-learning algorithm. This project explores the ability of a robotic hexapod agent to learn how to walk, using only the ability to move its legs and tell if it is moving forward. Thus, the hexapod may be seen as an analog for a biological subject lacking all but the basic instincts observed in infants and having no external support or parental figure to learn from. The problem is approached from the perspective of exhaustive experimentation, which is simplified to ensure that the agent learns an acceptable walking gait in a relatively short time. This method results in a hexapod capable of walking forward with some efficiency and continuing to learn as it exploits its own actions. After implementation, the robot learns how to walk.

Index Terms—Reinforcement Learning, Cognition, Control Theory, Robotics

I. INTRODUCTION

The problem of walking is perhaps different for any configuration of the legs, from centipedes and spiders to bipedal humans and apes. The hexapod project is a case between those, concerning a hexapod most similar to a large insect. The problem of a very young creature learning to walk is common in nature and the project explores the issue through the use of a robotic agent which learns without outside influence. Much is to be learned from the experimentation of learning rates and effective strategies for problem solving in a creature's brain by exploring the means by which a mechanical facsimile can do the same. In this case, we will be using Q learning to exhaustively traverse the possible moves made by the legs of the hexapod.

The primary problem faced by Q learning is one of time. While a computer is capable of handling and redistributing the memory requirements of testing eighteen motors at high degrees of precision in every possible configuration, the time required would be on the order of months of testing. Through simplification, positions are reduced for each leg to a

combination of motor configurations producing three positions per leg; this pared the states down to 729 total states. Further reductions used analysis to determine opposing leg motions which would not produce any form of forward movement, eliminating 96.11% of the state-action matrix from consideration.

Another obstacle was the learning algorithm itself. Initially there was an exhaustive approach which would be guaranteed to cover every single action state, however this process did not allow for continual learning while moving in exploitation mode (which was explicitly separate from exploration or 'learning' mode) and partial results would not preferentially favor states likely to be in an efficient walking gait. The strategy was instead shifted to a new method, which would randomly seed actions, but statistically favor the states with known useful actions. The result was a quick path to a workable walking gait, which improved and adapted over time even during exploitation mode.

II. MATERIALS AND METHODS

A. Apparatus

The coding of the project took place in two languages, using two separate programs as the basis for the code. Control and application of the theory is done through the use of MATLAB. MATLAB is a very powerful yet intuitive program to use, and works very well with matrix manipulation. The other base program was an OpenCV motion tracking system furnished by Kiwon Sohn from the University of Arizona. Using file IO commands to interface the two programs, Motion tracking code fed motion information to MATLAB while MATLAB controlled the timing of the camera. The robot ran at full speed slightly more than two actions per second at peak, slowing somewhat when computer memory filled up.

The robot used to implement Q-learning is a hexapod robot with six legs. Five of the six legs are similar, whereas one has been upgraded with a stronger motor. Eventually it would be desired to have all six legs be identical with stronger motors, however due to budget constraints only one leg can have an upgrade. Each leg has three degrees of freedom. The coxa joint controls the horizontal angle of the leg to the hexapod. Each leg also has a femur and tibia, as mimicked from biological systems. For the five matching legs, a Futaba s3003 ball-bearing servo motor controls the coxa. Dynamixel AX-12 servo motors are used to position the angle of the femur. These motors were originally intended to be used to get torque feedback for efficient walking gaits, however for the scope of Q-learning implementation, other simpler sensors were incorporated. Futaba s3305 servo motors adjust the

Manuscript received May 11, 2009.

M. R. Bunting is with the University of Arizona, Tucson, AZ 85705 USA (e-mail: mosfet@email.arizona.edu).

J. Rogers, is with the University of Arizona, Tucson, AZ 85705 USA (e-mail: PsiVen@email.arizona.edu).

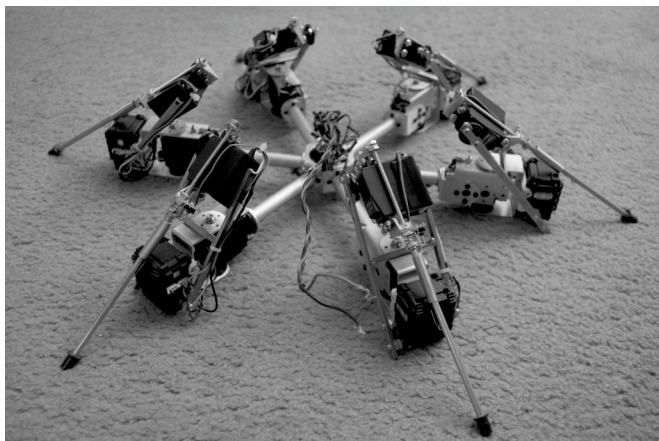


Fig. 1. Robot hexapod platform. Each leg has three degrees of freedom. Futaba s3003 motors operate each coxa, Dynamixel AX-12 motors operate each femur, and Futaba s3305 motors operate each tibia. Push buttons are mounted on each tibia to sense ground contact. Shown with no electronics.

angle between the femur and the tibia. The sixth leg incorporates a Dynamixel RX-28 servo motor to adjust the femur angle, and an AX-12 servo motor adjusts the angle between the femur and the tibia. Each leg is located circularly symmetrically from one another. To position each motor, inverse kinematics has been implemented to make state selection easier. A picture of the bare hexapod with no electronics can be seen in figure 1. The standard Futaba servo motors are controlled through a Pololu serial servo controller. A custom programmed PIC18F4550 microcontroller pipes information from a Bluetooth serial module to both the Dynamixel half-duplex bus as well as the Pololu controller.

For reward measurement, two sensors are used. On each leg, a push button has been added to each tibia. This way, each leg can be checked to see if there is contact with the ground. There are many ways to convert the six buttons into a partial reward. The simplest method is to give a reward if more tibias are touching the ground. If there is only a small number buttons pressed, then only a small number of legs are supporting the hexapod, resulting in a cost instead of a reward. The buttons are located on the tibia below the tibia motors, as seen in figure 1 and 2.

The second sensor is the main sensor of hexapod. Since the goal is to learn the simple task of walking forward, a sense of forward movement is needed. A Quickcam Communicate Deluxe is mounted on the front of the hexapod. From this, optical flow can be measured to get a sense of movement. In order to get a better sense of orientation, a cheap door peephole was disassembled and attached to the camera to get a wider field of view. The camera with attached peephole is shown in figure 2.

As stated earlier, OpenCV is used to calculate optical flow. When the robot is in one state, MATLAB communicates to the optical flow program and triggers the program to take a frame. The robot then takes an action, moving into a new state. From this new state, MATLAB again triggers the optical flow program to take another frame. From these two frames, the optical flow program calculates the optical flow using the Lucas-Kanade method. Before information is sent back to MATLAB, the optical flow program crunches part of the data.



Fig. 2. Quickcam Communicate Deluxe web cam used for vision. A door peephole was disassembled and quickly mounted to greatly improve the field of view, resulting in better orientation measurements from optical flow.

Optical flow vectors have been drawn on an image as seen in figure 3. This image is generated from the optical flow program. The program determines the magnitude of each vector, then takes the overall average and standard deviation. Optical flow vectors that are beyond a standard deviation from the mean are not considered in any of the following calculations. The vectors that are considered are drawn in green. Vectors that are not considered are drawn in red. Clearly the red vectors are a result of improper feature matching between the two images from the optical flow function, and hence the data is thrown out.

From the good data, there are three main motions that can be easily measured. The first motion is translational movement. This is determined by averaging all of the x and y components of the optical flow vectors. Originally there was minimal distortion of the image since no wide-angle lens was incorporated. Since translational movement is not a strongly desirable motion, distortion was not accounted for in the code. Translation between two states could be considered as a cost.

The second part that can be measured is the forward movement. The white X in figure 3 is the center of the image. First, a location vector is determined from the X to the beginning of the optical flow vector. This vector is then normalized. A dot product is taken between the location normal vector and the optical flow vector. This is repeated for all considered vectors in the image. All dot products are then averaged. If the hexapod were to move forward, the vectors would be mostly pointing outward from the center of the image, resulting in a higher result from the dot product. If the hexapod were to move backward, the vectors would be mostly pointing towards the center of the image, resulting in a lower negative result from the dot product. These values can clearly be directly implemented as a reward.

The third motion to be measured is tilt. Again, the location normal vector is determined. When the hexapod tilts, vectors close to the center show little movement, while vectors far from the center show larger movement. Because of this, each optical flow vector is then divided by the magnitude of the location vector, resulting in a compressed optical flow vector. A cross product is then determined between the normalized

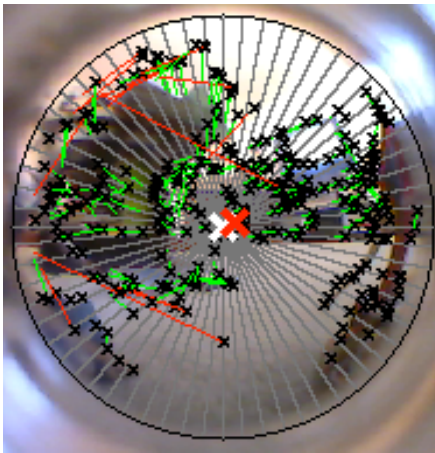


Fig. 3. Image from camera. Vectors are drawn from optical flow calculation. Green vectors are used in orientation calculation. Red vectors are beyond a standard deviation of the vector magnitudes, and are not used in orientation calculation. 64 different sectors of image are shown for optical flow vector weighting.

location vector and the compressed optical flow vector. Again, this is repeated for all optical flow vectors, and the resulting average of all cross products is determined. Since tilt is undesirable, the result is considered to be a cost when the magnitude of the average is higher.

One problem with using the OpenCV optical flow calculator is that the optical flow vectors are determined by finding the best features in one image, then finds the same features in the second image. Fortunately, this generally results in better optical flow vectors. The problem is that if there are an excess number of features in an area of the image, then the previously described calculations can give poor representations of the corresponding motions. Ideally, the best data to calculate motion would be uniformly distributed optical flow vectors over the image. If there are mostly vectors located in the top of the image, and the hexapod made a purely translational movement downward, then the translation data would be fairly correct, however the dot product, forward movement result would be very high, resulting in highly inaccurate data. The image in figure 3 shows that the floor had very little features, resulting in no vectors in most of the areas. The top left however shows a number of features, opposite to the floor. To compensate for this, sixty-four different sectors are considered. The code then counts the number of vectors in sector. A histogram is built from this data, as shown in figure 4. The different sectors are drawn in figure 3.

From the corresponding number of features in each sector, different weightings are imposed on the vectors in the sector. These weightings are determined before the three different motion calculations. Weightings are determined by comparing the number of vectors located in one sector against the number of vectors in the opposite sectors. If there are five vectors in one sector, and three vectors in the opposite sector, then the five vectors are multiplied by a three-fifths. If there are no vectors in another sector, and there are ten vectors in the opposite sector, then the ten vectors are multiplied by zero. After the determined weighting, each motion is then calculated considering the corresponding weights. This method resulted in cleaner data considering the non-uniformly distributed

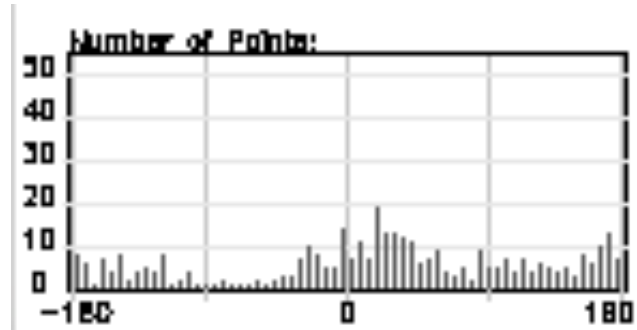


Fig. 4. Histogram from image in figure 3. Oppositely located sectors are compared to weight the corresponding vectors to obtain better movement information.

optical flow vectors.

Once the optical flow program has determined a general sense of motion, data is then piped to MATLAB. From the three general motions and the button information, a reward system can be built in MATLAB.

B. Methods

In simple Q-learning, a finite number of states must be present. Due to the large number of degrees of freedom, the total number of states and actions to be taken becomes too large to consider. Due to this, the smallest number of states for the hexapod to walk will be incorporated. The minimum number of states need for forward motion is three states per leg. The zero state is the lifted state. The other states place the feet of the hexapod below the body, such that they are parallel to the direction of motion. State one places the foot closer to the front of the hexapod. State two places the foot closer to the back of the hexapod. Since there are three states per leg and six total legs, this results in 729 possible states.

From every state, the hexapod can take an action. Due to the simplicity of the states imposed, from any single state, the hexapod can move directly to any other state. This implies that there is an equal number of actions as there are states.

For Q-learning, The Q values are store in a Q matrix. The conventional, simple Q-learning calculation is shown in equation 1, where s is the current state, a is the action taken from the state, R is the reward from the state-action transition, γ is a discount factor less than one, and s' and a' are then next action and states. Essentially, the Q value is based upon the reward for taking an action, plus the maximum Q value in the next following action.

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(s',a') \quad (1)$$

Q is initially filled with arbitrary values. For the hexapod, Q is filled with zeros. The hexapod then is placed into an arbitrary state. For the project, the hexapod is initialized in the zero state, so that all legs are in the lifted state. This state is called resting state. From these initial conditions, a programmed loop begins. First, an action is chosen. Choosing an action can be either intelligent or purely random. Once the hexapod performs an action, equation 1 can be used to store the Q-value. The action taken is now the new state.

The process then repeats for a chosen number of iterations.

For the scope of this project, the Q values are determined as in equation 2, as taken from Barto and Sutton. Rewriting the equation, it becomes clear that their form of Q-learning is nothing more than a Kalman filtered version of the conventional Q-learning.

$$Q(s,a) = Q(s,a) + \alpha [R(s,a) + \gamma \max_{a'} Q(s',a') - Q(s,a)] \quad (2)$$

$$Q(s,a) = \alpha [R(s,a) + \gamma \max_{a'} Q(s',a')] + (1 - \alpha)Q(s,a) \quad (3)$$

The Q matrix is $s \times a$ in size. From this, we can calculate the number of points in the Q matrix, as in equation 3. Clearly even with only the minimal number of required states per leg, the total number of values to explore is rather large.

$$\text{size}(Q(s,a)) = 729^2 = 531441 \quad (4)$$

Since the measure rewards are not in simulation, a large amount of time is needed to perform each action as compared to a computer simulation. Even operating at a fairly high speed of one action per second, the hexapod would require six days of continuous exploration to take every possible action. Also, even after exploring all actions, Q-learning requires many more iterations to build an efficient Q matrix, due to its recursive nature. This means that many more iterations are required to reach approximate convergence.

From this, a large limitation of Q-learning becomes apparent. Even with the heavily constrained system of having only three states per leg, having the hexapod explore all possibilities of the simplistic system still requires an enormous amount of time. It becomes appealing to incorporate a more intelligent form of action selection. Softmax action selection provides a nice balance between learning exploration versus exploitation. A probability distribution is built for all possible actions, as in equation 5, where τ is called the temperature.

$$\Pr(a) = \frac{e^{\frac{Q(a)}{\tau}}}{\sum_b e^{\frac{Q(b)}{\tau}}} \quad (5)$$

The higher the temperature value, the more uniform the distribution, resulting in a more explorative behavior. The lower the temperature, the less uniform the distribution, resulting in a more exploitative behavior. Choosing the temperature almost becomes more of an art than a science, depending on the programmer's desire. Once the distribution is built, a random number is chosen which decides a corresponding action.

Considering the amount of states, it becomes desired to immediately remove any possible actions that clearly will not result in a reward. Also, some actions can very easily strain the inexpensive coxa motors. For example, if leg 1 is in state two, and leg 2 is in state 2, and the action results in leg 1 in state 2 and leg 2 in state 1, then the opposing motion could result in unnatural torque on the leg components. Also, if there are any two adjacent legs in the zero state, then there is a good chance that the robot could fall over. Considering all of the undesirable states and actions, a quick simulation can eliminate all state-actions that do not need to be measured.

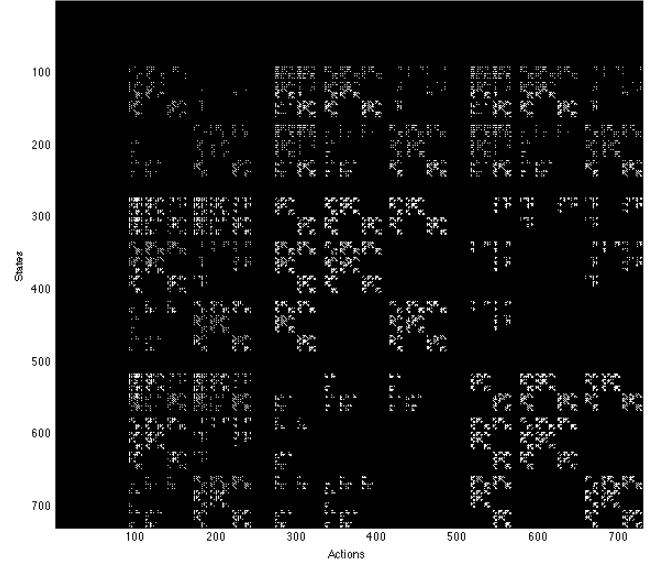


Fig. 5. State-actions simulated that are not to be tested. Black areas show states and actions never to be tested. White areas show good state-action transitions. 96.11% of state-actions have been eliminated.

State-actions that have been eliminated are shown in figure 5. This results in 96.11% of state-actions that do not need to be explored.

Considering all of the above initial conditions, it becomes clear that learning will take a much shorter amount of time. The last major component of software to consider is the reward value. If there are less than three tibias in contact with the ground, then the reward becomes -10. If there is no tilt or translational movement, and more than two tibias are in contact with the ground, then the reward becomes the forward value from the optical flow code. If there are more than two tibias in contact with the ground, and there is translational or tilt motion, then the reward is calculated as in equation 6.

$$R(s,a) = \frac{\text{forward}}{\sqrt{\text{translation}^2 + \text{tilt}^2}} \quad (6)$$

III. RESULTS

After the implementation of the constraints, the hexapod was initialized in the resting state. From the resting state, the hexapod implemented the softmax decision process to build the reward matrix. After ten thousand measured points, it became clear that since off-board processing was going to be implemented, it would be possible to have the hexapod purely explore. This was implemented by setting unexplored Q-values very high. Once the reward matrix would be completed, then off-board processing could be used to build the most efficient walking gait from the measured, now static reward values. Off-board processing would run the softmax action decision process under many iterations, resulting in a fairly well established Q matrix. The full original program could then run again with the Q matrix, such that new rewards could be measured in case previous measurements were noisy or simply invalid. For example, since only one measurement was taken for an action, this could have resulted in a negative reward from the optical flow, when truly the result should have been a positive reward. By running the hexapod with the

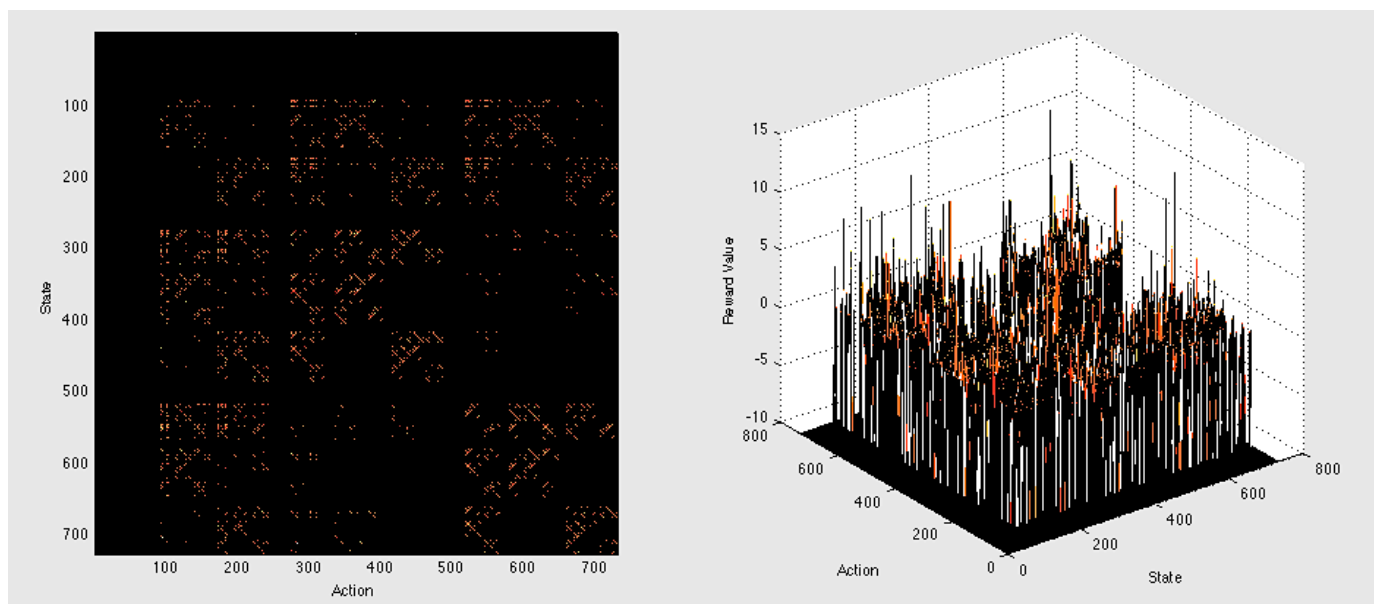


Fig. 6. Resulting reward matrix. Data measured from taking an action from each state. Matrix as shown is 99.65% complete, with 1872 points to be measured.

calculated Q matrix, the rewards would be reconditioned and therefore the Q values would be re-determined, resulting in a more conditioned walking gait.

Unfortunately, during the reward gathering process, the Pololu serial servo controller burned out, resulting in the lack of operation of eleven servo motors. This resulted in the inability to proceed further with reward measurement. Fortunately however, only 1872 data points were left unmeasured, so the reward matrix was only 99.65% complete. Since no conditioning process could be implemented, the only part that could be tested was to see if an efficient walking gait could result from off-board Q-value processing. Figure 6 shows the resulting reward matrix. The darker areas are smaller rewards, and the areas more red are higher rewards.

With the almost complete reward matrix, off-board processing implemented the softmax action selection to build the Q-matrix. For the parameters, τ was set to 15, α was set to 0.8, and γ was set to .95. The reason for a large discount factor was so that future actions would be weighted higher. This helps ensure that a good walking loop in the Q matrix could be entered as quickly as possible from any state. Under these chosen constants, different numbers of iterations were performed in the Q calculation. Four of the resulting Q matrices can be seen in figure 7. At first, a thousand iterations was clearly too small to build any Q matrix. After then-thousand trials, the Q matrix begins to build its shape. When one hundred thousand iterations were performed, Q appears to converge more towards the case of a hundred million iterations.

Because of the resulting Q matrices, it was chosen to run the same number of iterations as there were elements in Q. After running the iterations to build the Q matrix, a simple exploitation program was built to exploit the Q matrix. The resting state was the initial state, and the action with the highest Q value was chose for each step. The resulting output from the exploitation can be seen in figure 8. Notice that after

a bit of confusion from the initial state, only five steps are needed till an efficient walking gait is found and repeated.

IV. CONCLUSION

Q-learning is clearly an effective method for learning when considering small state-action spaces. A hexapod with a minimum of 1024-bit resolution per motor clearly has an unconstrained state-action space that is too large for the simplicity of Q-learning. Though the resulting Q matrix from the off-board processing clearly came up with a four step walking gait, there was some uncertainty in the beginning. It is strongly desired to incorporate this Q matrix into the hexapod after it is repaired to see if further conditioning could occur, or if a three step walking gait could be discovered.

One other simple possible incorporation to the project is to build a separate Q matrix that receives better rewards upon having a net backward movement. This way, both forward and backward learning can take place at the same time. Also, adding two more sates per leg could include the possibility of having turning capabilities, resulting in better environment exploration.

In terms of gait synthesis of the hexapod, it would become very interesting if more advanced learning techniques could be implemented in the hexapod. This way the robot would not be constrained to a small state-action space. Other learning techniques such as function approximation become very appealing in this aspect.

REFERENCES

- [1] R. S. Sutton and A. G. Barto, *Reinforcement Learning, An Introduction*. Massachusetts: The MIT Press, 1998
- [2] R. T. Sikora, "Learning Optimal Paramete Values in Dynamic Environment: An Experiment with Softmax Reinforcement Learning Algorithm," Available: <http://ieweb.uta.edu/vchen/AIDM/AIDM-Sikora.pdf> [Accessed: April 27, 2009]
- [3] L. P. Kaelbling, M. L. Littman and A. W. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, vol.4, 1996

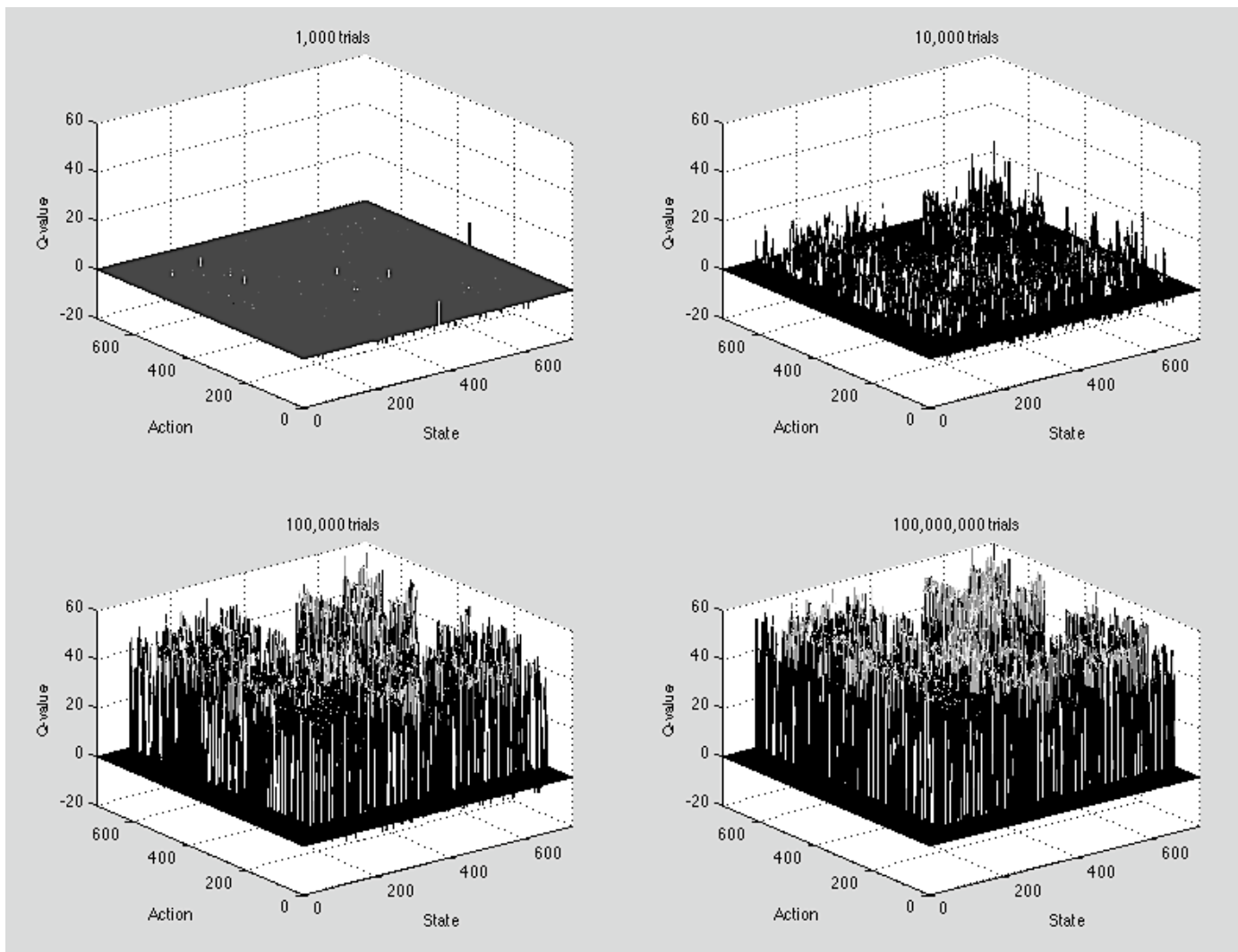


Fig. 7. Q-values under different numbers of iterations. Between 1,000 and 100,000 trials, the Q-values change dramatically. Between 100,000 and 100,000,000 trials, the Q-values change very little, showing approximate convergence.

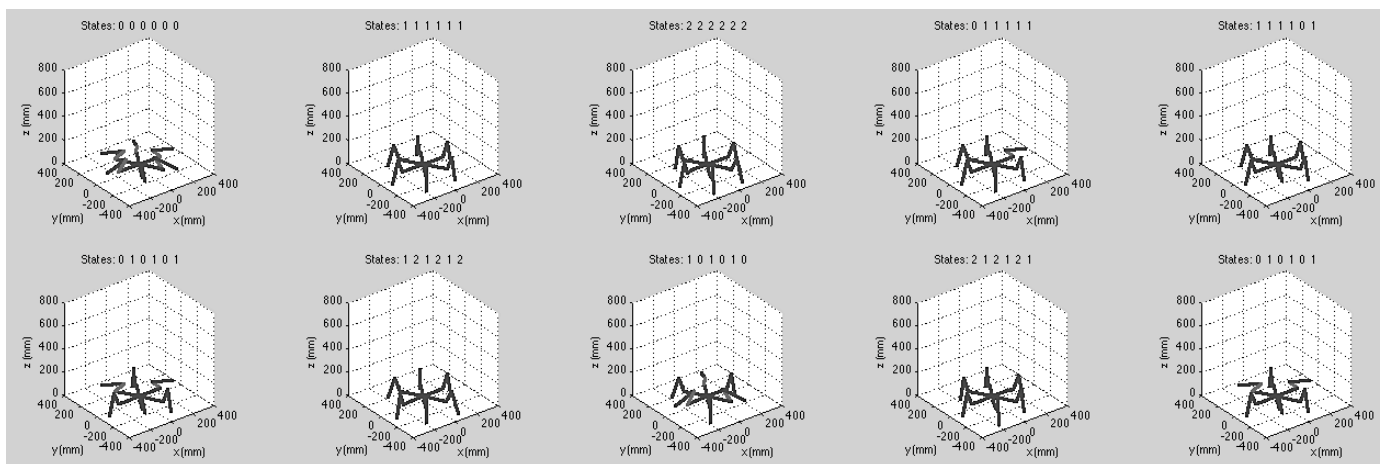


Fig. 8. Complete, learned walking gait. Each step is the next maximum Q value action, with walking in the positive y direction. The hexapod begins from rest state (all legs up, top left) then takes the best Q value resulting action to enter the next state (going to the right). The hexapod performs some non-ideal actions till the sixth state (bottom left) is reached. From this action, only four actions are needed to perform a fairly ideal walking gait, until the first sixth state is repeated. The sixth state and the tenth state (bottom right) are identical, and hence the following actions and states are purely repetitions of states and actions six through nine. Individual leg states are shown above each drawing.