Using Neural Networks to navigate "Car on the Hill" on-line

Leonid Kuvayev

January 4, 1996

1 Introduction

Researchers in the Reinforcement Learning community tend to pay close attention to TD and Q-learning methods. These methods demonstrate a high potential for learning a broad class of control and reinforcement learning problems. We are interested in the paradigms where the rewards and costs of actions are known and the goal is to maximize the total rewards. This generality distinguishes the RL methods from search and planning methods. In the real world there is a rare chance of knowing apriori the goal location however the reward (e.g. food) is easily sensed once the goal is reached.

In the past the research went in the various directions. Many new algorithms were invented including Dynamic Programming, Prioritized Sweeping, Sarsa, and Q-Learning. We can divide them in two broad categories: off-line and on-line methods. The strict convergence theorems were proved for the off-line dynamic programming in (Bellman, 57). However the convergence results for on-line learning are much weaker. In fact, the on-line methods using function approximation do not work for certain problems. The difficult issue in the on-line class is a tradeoff between exploration and exploitation policies.

The "Car-On-Hill" (or COH) problem serves as a testbed for this project. The tabular and neural-network function approximation will be compared on the basis of Q-learning.

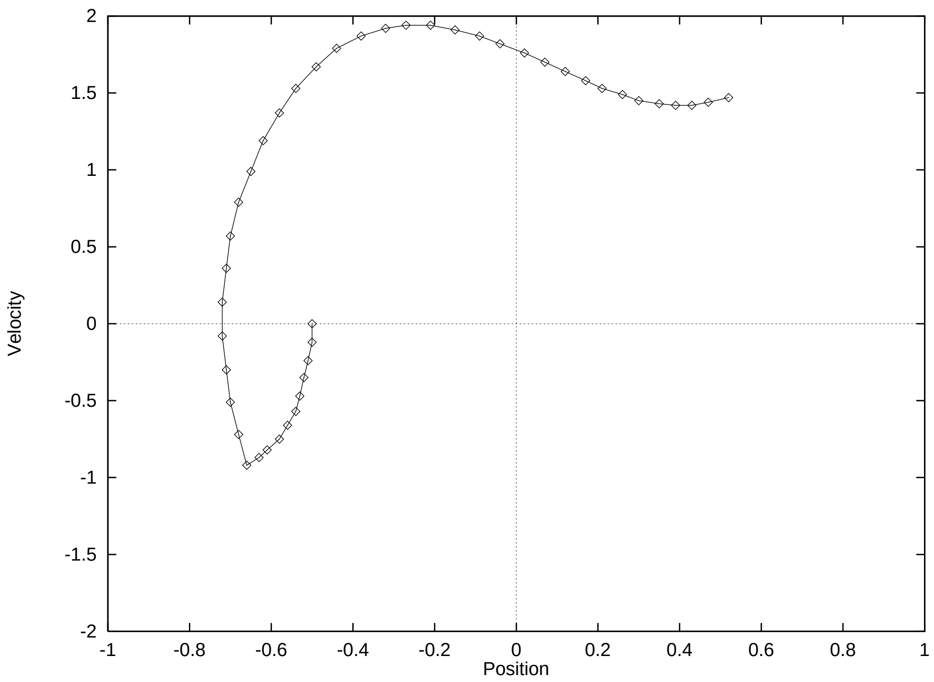

Figure 1: Optimal path.

2 Car On the Hill

In this task a car starts at the bottom of the hill and is supposed to climb to the top. The car can execute one of the three actions at a time. It can accelerate forward, backward or maintain the current speed. The hill is too steep for the car to overcome in a straight run. The optimal policy is to go backwards to gain some momentum first and then rush forward while keeping on the acceleration. Figure 1 shows one of the optimal traces of the car. To simulate the movements of the car the time was discretized. For each small time increment a new acceleration, velocity and position of the car are calculated. The hill dynamics were obtained from Justin Boyan at CMU.

The motivation behind the choice of the problem for this project stems from Boyan's negative results for off-line learning using neural networks. Our goal is more challenging and is to show that the neural network can learn during on-line policy. On-line policy is preferred because most of the real world problems provide no model of the environment.

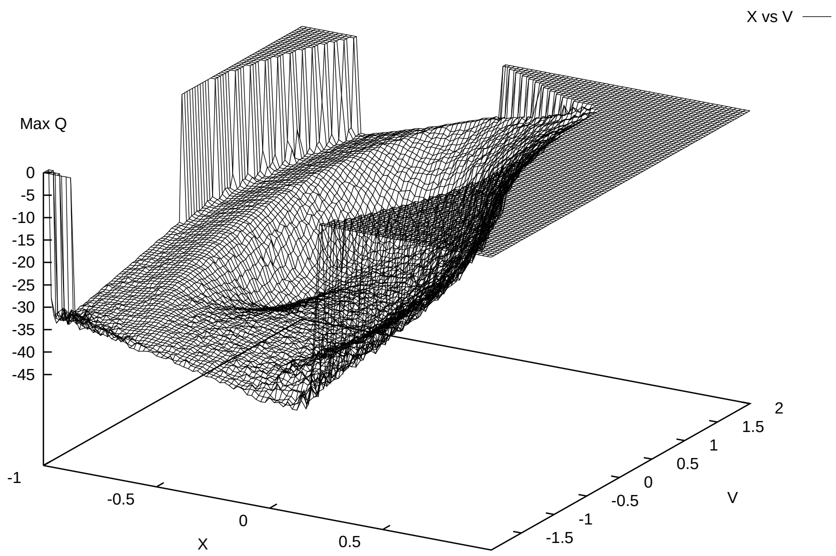

Figure 2: Tabular surface.

The value surface for this problem is harder to approximate than the surfaces for the maze problems. It can be noticed from Figure 2 that the surface has various steepness. The values drop dramatically if the wrong action is performed in the goal proximity. To illustrate this one can imagine how accelerating backward on the steep incline will decrease the car's momentum and will make it to slide backwards and to start over again.

3 Q-Learning Algorithm

The algorithm used to learn COH problem is a traditional Q-learning algorithm proposed by (Watkins, 89). The reward of 0 is given when the car reaches the top and a trial terminates. For each step the car receives the punishment of -1. This is an instance of minimal time problems. The Q-values represent the utility of an action and approximate the remaining number of steps to goal if this action is taken. The trial starts at the bottom of the hill with $x = -0.5$ and $v = 0.0$. Ten thousand trials constitute an epoch. Several epochs can be run to obtain smooth learning curves. The algorithm consists of the following steps:

1. Initialize Q-values.
2. Until the goal ($x > 0.5$) is reached, repeat
   (a) select the action a using *SoftMax*
   (b) move the car by applying action a
   (c) update the old Q-value
   $Q(s_{old}, a) := \alpha(\max_a Q(s_{new}, a) - 1) + (1 - \alpha)Q(s_{old}, a)$

In part 1 the initial Q-values vary with the type of the function approximator. *SoftMax* is an exploration policy in which the action probabilities have the Boltzman distribution, i.e. $a = A_i$ with $Prob = e^{Q_{A_i}}/(e^{Q_{A_0}} + e^{Q_{A_1}} + e^{Q_{A_2}})$ where $i = 0, 1, 2$. In this task A_0, A_1, A_2 are backward, neutral, and forward actions.

On-line Q-learning algorithm does not require a model of the environment. The Q-values are updated when the agent passes through corresponding states. The convergence is proven in (Watkins, 89) for the Q-values that are on the valid trajectory. Modeling real world problems requires extensive or even infeasible computations. The model-free property of Q-learning makes it preferable over the value-iteration algorithms, such as AHC (Sutton, 90).

4 Tabular Experiments

To test the correctness of the Q-learning algorithm the experiments were performed using tabular function approximation. This is the robust method for which the convergence is guaranteed. The Q-values are initialized to 0, same as the goal value to promote the exploration of new states.

The learning curves are shown on Figure 3. They were averaged over 10 epochs. Four graphs show the influence of various values for learning rate α. As expected a higher rate make the convergence faster. Under the high rates ($\alpha > 0.5$) the greedy policy is unstable and may not be optimal. The

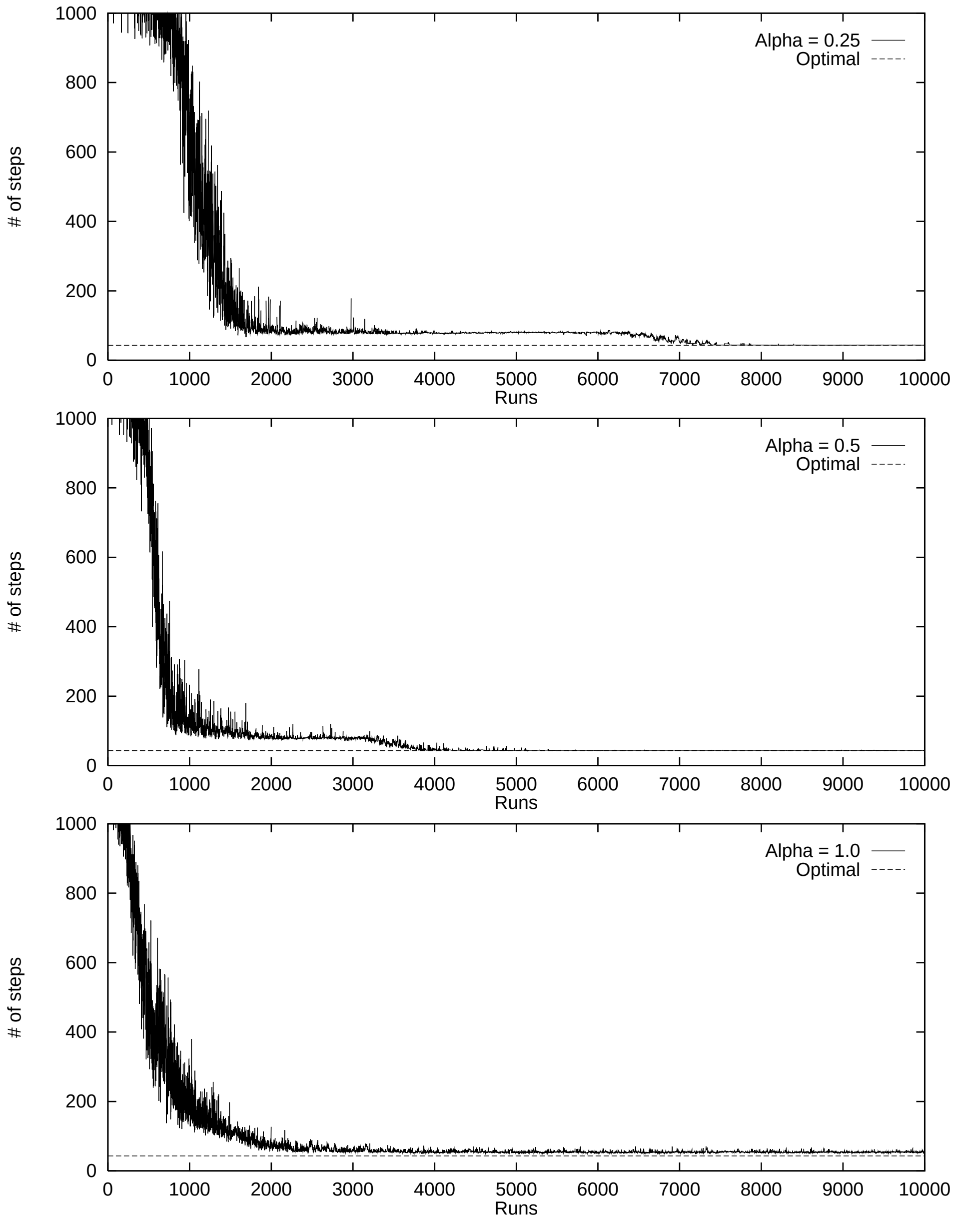

Figure 3: Tabular Q-Learning.

best rates are the intermediate ones. For example with $\alpha = 0.5$ the learning occurs after 4000 trials and the policy is within 5% from optimal. The surface comprised of the highest Q-values is shown on Figure 2. It is possible to see the spiral trajectory that starts at the bottom and rises smoothly to the 0-level plateau. Two abrupt 0-level corners indicate that these states were not explored and maintained their initial 0 values.

5 Neural Network Experiments

Using Neural Networks as a function approximator is often a challenging task in RL framework. The problem with neural networks is that they generalize the values of not visited states. Their function approximating and generalizing qualities are useful on the smooth surfaces where missing values can be interpolated from the nearby values. However in the case of the jagged surface as in the COH domain the neural network does a lot of harm by substituting the unseen optimistic values with the pessimistic estimates of the neighbors. This is an especially acute problem during the first trials. In the COH there is no chance to stumble on the goal state by chance because of the long distance to the goal (43 steps) and the narrow solution corridor. The only way the goal may be found is by exploring the new states. This approach works perfectly with the look-up tables. The values of visited states are driven down by step cost of -1 while the unseen values remain to be 0. The neural network receives negative reinforcement for a long time so that the values for the unseen states are approximated and become in unison with their neighbors' pessimism. They do not look appealing to the agent anymore, who instead goes in circles depressing all values even further. If left on its own the agent's estimation of Q-values is linearly decreased to $-\infty$.

To help the agent during the initial stage of learning the guided trials are used. The idea is to allow the agent to receive the final reward of 0 and increase the value of the actions that led to the goal. It is done by running the agent through a guided trial where the actions are chosen by a fixed successful policy. The selected policy is not optimal so it takes an agent 117 steps instead of 43 steps to reach the goal. It was done intentionally to test if the agent comes up with improved (or even optimal policy) or if it adopts the given one. The policy makes the car to swing forward and backward until enough momentum is gained to overcome the hill. The car does the

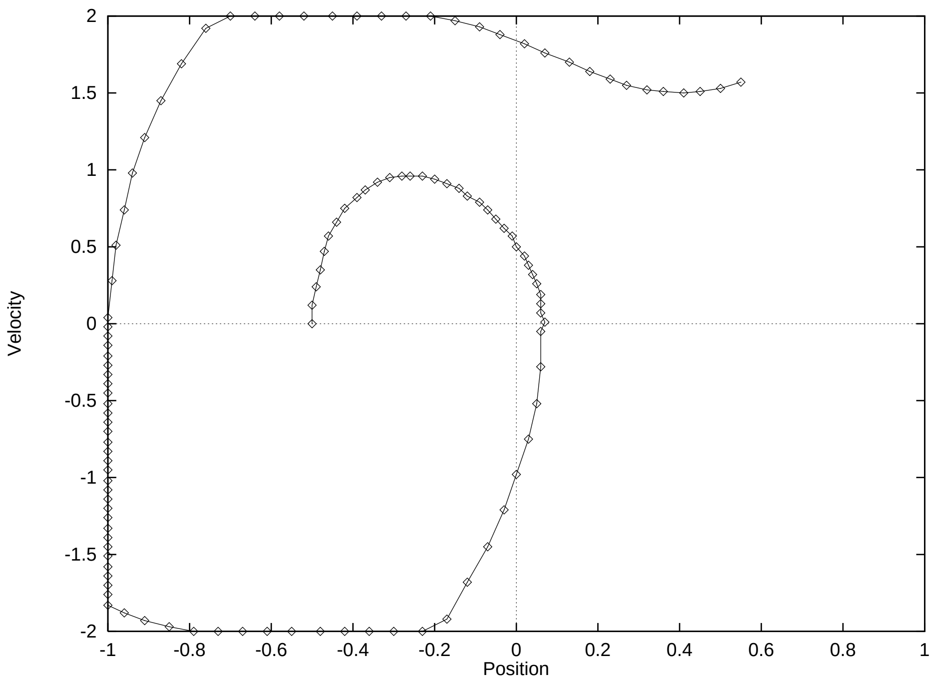

Figure 4: Guided path.

extra swing due to the wrong initial direction as shown on Figure 4. The question remains how to incorporate the guided trials and still continue to explore. After each exploratory trial the greedy test run is performed on the Q-values. If the agent finds the solution within the fixed number of steps (200) then it probably "knows" what it is doing. If the greedy policy does not lead to the goal it means that the agent goes in circles and needs help. In such case at the next trial the actions are provided by described above policy. As soon as the agent learns to finish the run, the exploratory trials replace the guided ones. Figure 5 shows that guided trials are necessary only during the first 500 runs, after that the surface is shaped sufficiently to allow the agent to reach the goal by following the greedy policy. Exploratory trials ensure that the agent tries out the reasonable actions and eventually finds the shortest path. The neural network architecture consists of three identical neural networks, one for each action. Each neural network is a feed-forward fully connected net with three layers. There are only raw inputs (position and velocity) used. The third input unit is a bias. The hidden layer required

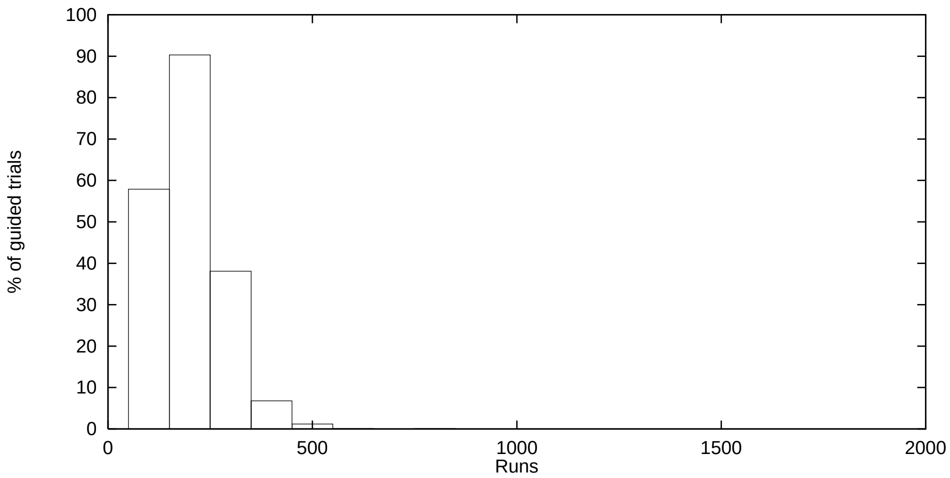

Figure 5: The percentage of guided trials.

a substantial number of hidden units (25). The more compact representation hindered learning. The output is a single unit that approximates the utility of the particular action.

Initially the experiments were run with the fixed small learning rate α. This rate is not the one that was used in the tabular experiments but plays the same role. The neural network receives the reinforcement through the squared error $\mathcal{E} = \max_a Q(s_{new}, a) - 1 - Q(s_{old}, a)$. There is no discounting associated with the error, instead all the weights are adjusted by the amount $w_{old} = w_{old} + \alpha \Delta w$. This parameter α adjusts the convergence speed in the similar fashion to the tabular α. The learning rate for the neural networks should be much smaller than the one for the table. The $\alpha = 0.01$ is sufficient for the convergence of the Q-learning algorithm. Figure 6 displays the length of the paths during training and testing periods. The convergence is not quite stable despite the 10 epochs averaging.

To further decrease the deviations from the optimal policy the learning rate was gradually reduced from 0.01 to 0.00006 by multiplying it by 0.95 every 100 trials. Figure 7 demonstrates the stable performance during testing without slowing down the learning. In both cases the learning was completed after 100 trials. Afterwards the weights should not be changed significantly in order to guarantee complete convergence. The effect of the learning rate on the neural network is much more drastic than on the look-up table. The

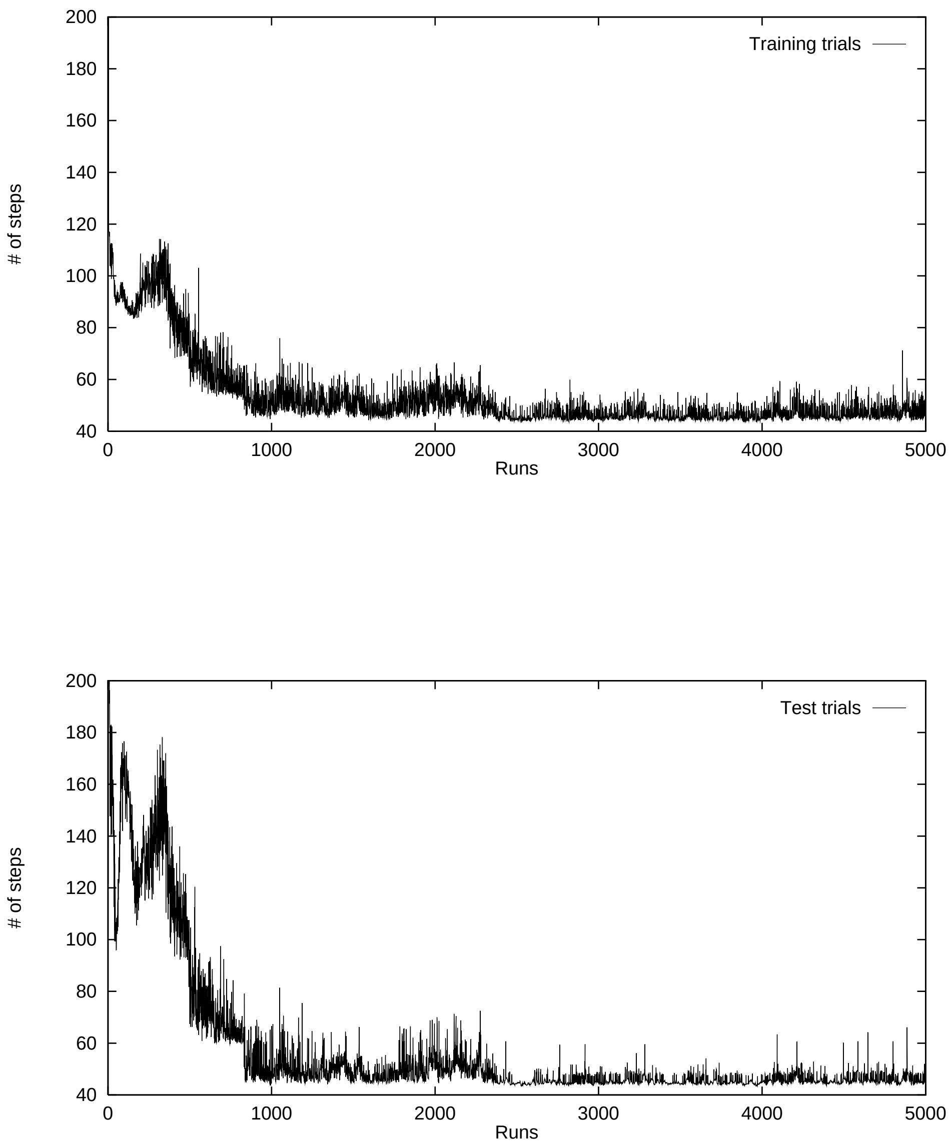

Figure 6: Neural Net Q-Learning (fixed α).

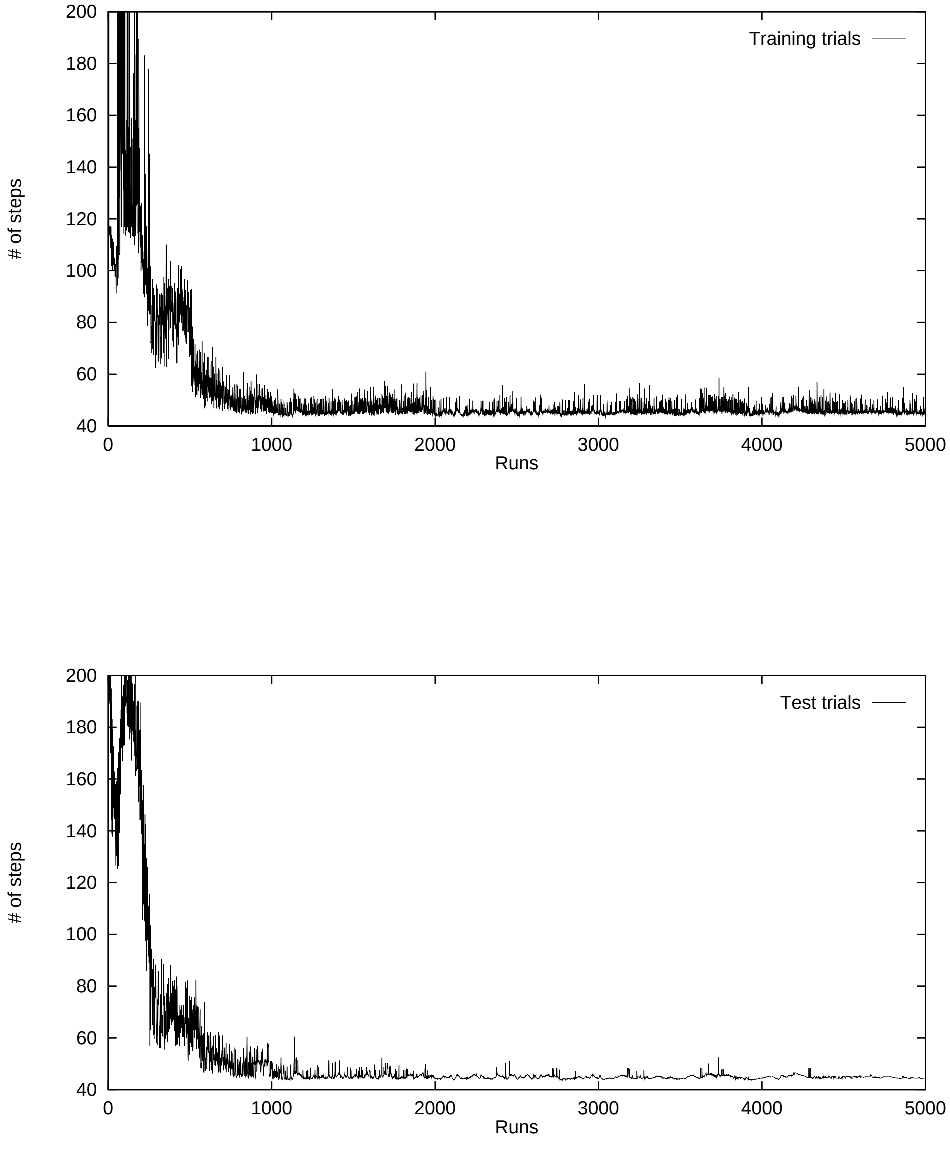

Figure 7: Neural Net Q-Learning (decreasing α).

values in the table may vary slightly while their relative altitude remains the same. The weight changes in the network may turn valleys into hills and vice versa.

Using guidance speeds up learning but creates a dependency on the external source of knowledge. It would be convenient if the learning system used it once and then used the currently best policy instead of the external guidance. Experiments were run to see if the improving policy converges faster than the fixed guided policy. One would expect the improving policy to be a more accurate advisor. However the COH properties prove the opposite. The fixed policy had 100% convergence results while the constantly improving policy resulted sometimes in “unlearning”. When converged the improving policy resulted in a slightly slower and a less stable performance. To explain this phenomenon we imagine that the fixed 117 steps policy provides a foundation on which the better policies are local peaks. One of them is a global maximum with 43 steps to the goal. If the agent is guided to climb a local peak than it has to unlearn this policy in order to reach the global maximum. In case of neural networks it may result in a catastrophe. Thus the fixed bad policy provides a convenient starting plateau for the neural network climber.

6 Conclusions

In this project we investigated learning with neural networks on the example of “Car On the Hill” task. Neural networks are able to approximate complex non-linear functions and can be applied as function approximators in Q-learning algorithm. Their advantage over the look-up tables are the low memory requirements and simple use of raw inputs. Their convergence speed is comparable to the one of the look-up tables. On the negative side are their vulnerability from the learning rate and the dependency on the external advisor.

The main contribution of this project is the mechanism for the incorporating the external advice into the neural network driven, on-line learning. The COH was considered too hard for the neural networks to learn by Justin Boyan at CMU who showed the divergence of the weights in a dynamic programming algorithm. The careful use of an advisor allows the stable convergence in the on-line case. To make a more confident statement further investigation is needed in the domains where neural network divergence is a problem. This work has been done independently of the results obtained by Long-Ji Lin who demonstrated speed-up in the learning through similar knowledge injections. We believe that on-line model-free case is more suitable for the real-time applications than AHC architectures (Sutton, 90). Nevertheless their comparison is necessary and will be a part of future research.

I would like to thank Nathan Sitkoff and Jeffrey Clouse for the helpful discussion and comments.

References

[1] Bellman, R.E. (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ.

[2] Lin, L.-J. (1992). Self-improving, reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8, 293-321.

[3] Rumelhart, D.E., Hinton, G.E., & Williams, R.J. (1986). Learning internal representations by error propagation. *Parallel distributed processing: Explorations in the microstructure of cognition. Vol. 1.* Bradford Books/MIT Press.

[4] Sutton, R.S. (1988). Learning to predict by methods of temporal differences. *Machine Learning*, 3, 9-44.

[5] Sutton, R.S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. *Proceedings of the Seventh Internation Workshop on Machine Learning* (pp. 216-224).

[6] Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning, 8*, 257-277.

[7] Watkins, C.J.C.H. (1989). Learning from Delayed Rewards. Ph.D. Thesis, King's College, Cambridge.