Evolving Adaptive Intelligence

Using NeuroEvolution with Temporal Difference Methods in the Game Domain

Nathaniel Tucker

Abstract

Adaptive intelligence is the ability of a system to respond quickly and effectively to changes in its environment during its lifetime. Responding quickly to change can be greatly beneficial in many domains, where unpredictable changes require generalizing adaptive behavior. Although there are many online reinforcement learning systems, manually selecting the proper representation can be challenging, and often leads to suboptimal solutions. By using evolution to automate the search for an appropriate representation, we are able to evolve agents better able to learn. We will examine the applicability of this concept by using real time NeuroEvolving Augmenting Topologies (rtNEAT) – an evolutionary algorithm for exploring the topological and synaptic weight space of neural networks – to evolve a function approximator for the Q-learning temporal difference method using an adaptive learning rate to more effectively adjust to dynamic conditions. This algorithm, real time NeuroEvolving Augmenting Topologies + Q-learning with Adaptive Learning Rate (rtNEAT+QwALR) will be compared in the dangerous foraging domain with adaptive assignments against rtNEAT, and rtNEAT+Q.

Introduction

In scenarios of competition, being able to detect an opponent's strategy and react accordingly can greatly improve performance. And in the case of game non-player characters, greatly improve the challenge and immersion. By choosing a strategy that best responds to the current conditions (the current opponent, or if the opponent changes behavior, the current strategy they have chosen), instead of simply relying on a general fixed strategy, weaknesses in the opponent's strategy can be exploited. Even if the opponents adapt to your strategy as well, this will force them into a sort of arms race of strategies that will at the very least not leave them exploiting your static behavior. One obstacle in achieving effective adaptation of opponent's behavior is being able to react to every possibility. If there are only a finite number of effective strategies an opponent can employ, then simply enumerating proper responses coupled with a detect and memory system would be enough. (Although this task can become challenging with an increasing number of strategies.) In addition, evolved solutions will often exploit unexpected invariants during training, which leads to a dependency on these invariants that does not model the more general conditions the agent must perform under. Furthermore, it may be hard or impossible to know all the possible adaptations that will need to occur so they can be incorporated into training, especially as the number of adaptive strategies increases. It would be desirable then, to be able to generalize adaptive behavior to be able most effectively respond to an infinite set of strategies, and also reduce the amount of preparation resources required (training).

Online learning mechanisms seem well suited for these problems, since they can update their behavior quickly within a lifetime. However, manually finding good representations is difficult and often results in a suboptimal solution. Evolution solves this problem by automating the search for good representations. By evolving a representation to be updated via online learning mechanisms, representations that are better able to learn are found – a trait that should lend itself well to the scenarios described above. To explore generalizing adaptive behavior, we will focus on combining the well established evolutionary algorithm rtNEAT with the online temporal difference method Q-learning. To respond more effectively to changing conditions, Q-learning will use an adaptive learning rate. We will use the dangerous foraging domain as a simple benchmark for basic adaptive behavior, and then a more complex competitive game environment will be used to observe this algorithm's ability to properly generalize the required adaptations in a more complex setting.

Background

I will first outline some previous work in this problem domain and similar domains. Following this, I will outline the algorithms used for my approach to this problem. I will assume a basic understanding of feed-forward neural networks, but I will build many other more basic ideas, such as backpropagation of error.

Previous results on adaptive intelligence

In an effort to cope with changing conditions in the lifetime of robotic agents, Floreano and Urzelai combined evolution and on-line learning techniques (1). However, instead of combining evolution with some preexisting online reinforcement learning algorithm, they worked on evolving the adaptive characteristics of the controller. Specifically, they encoded Hebbian learning rules in the genotype for each synapse, but not the synaptic weights. In a simple experiment with physical robotic agents, this learning algorithm with adaptive synapses (online learning) significantly outperformed the analogous fixed synapse algorithm. Furthermore, adaptive synapses were shown to be of great advantage against co-evolved prey. (1) Further experimentation revealed that these Plastic Neural Networks (the ones using Hebbian learning rules) (PPN) had the capability of learning and "…retaining behavioral abilities and displaying reinforcement-like properties." (2) The PPNs were also shown to outperform a form of recurrent neural network in time dependant tasks when sensory-motor re-adaptation is required (2).

In another attempt to achieve adaptive intelligence, plastic synapses (using Hebbian learning rules) were compared with solutions based on fixed-weight recurrent neural networks. Both were evolved using the NEAT algorithm. Surprisingly, the fixed-weight recurrent neural networks were able to solve the dangerous foraging domain faster and more reliably, although evolution was able to utilize adaptive synapses when they were available. (3) This suggests that there are domains that may require adaptive intelligence, but only require a

basic memory to solve. However, the optimal solution to this domain is simply to 'remember' which food was good, and then eat it when given the opportunity. The apparent requirement for adaptive behavior in this case, is really just a hidden Markov state, which is most accurately modeled by recurrent connections that provide a form of memory. Also, the amount of memory required is finite and small (only one memory for each food item). The real challenge for adaptive behavior is domains where changes are unpredictable and require a robust generalization of adaptive strategies, rather than simply a selection of the best strategy based on the revealed hidden state. It is in these cases that online modifications of a neural network should provide the most optimal solution.

Q-learning

Temporal difference methods have been shown to solve the reinforcement problem with good accuracy. Since temporal difference methods learn online, they are well suited to responding to unpredictable changes online. Specifically, Q-learning has been shown to produce good empirical results, especially when combined with neural networks as function approximators (4). Q-learning is able to compare the expected reward from actions without requiring an actual model of the domain. This alleviates the need to devote resources to understanding the domain in detail, reduces bugs derived from invalid assumptions, and enables solutions for domains that are intractable to understand.

The heart of the Q-learning algorithm is the Q function, which takes a state and action as input and returns the expected long term reinforcement from taking that action in the given state. By modeling the expected long term reinforcement, a policy that provides a proper balance of exploration vs exploitation can factor in all its choices, making the best one. Adding exploration to the otherwise obvious exploitation policy is essential to produce a more accurate Q function. Without an accurate Q function, the actions that could have significantly higher rewards than the current best action could go unnoticed and never be taken. In addition, without proper exploration, solutions are likely to get stuck at local max. With proper exploration, these actions can be discovered and then can be exploited during future states. (5)

To update the Q-learning function, the following update formula is used:

$$Q(s_t, a_t) \leftarrow (1 - \alpha_t(s_t, a_t)) \times Q(s_t, a_t) + \alpha_t(s_t, a_t) \times [r_{t+1} + \gamma \times \max_a Q(s_{t+1}, a)]$$

where t is time, $\alpha_t(s_t, a_t) \in \mathbb{R}[0, 1]$ is the learning rate, r_{t+1} is the reward at time t+1, $\gamma \in \mathbb{R}[0, 1]$ is the discount factor – which is a parameter that determines the relative importance of future reward to immediate reward. (5)

There are many ways to model the Q function, with the most obvious being a lookup table. This table has a cell for every state action pair that represents the expected reinforcement (the Q value). Unfortunately,

storing this table becomes intractable when the state and action space increase, since every combination of state-action pairs must have its own cell. Also, having distinct cells for every Q-value removes any ability to generalize, and likewise, requires significantly more training to fill all the cells. To alleviate these problems, function approximators are often used.

Backpropagation

Neural network function approximators have enjoyed good empirical success, alleviating these concerns (6). Using neural networks as function approximators for Q-learning involves setting the inputs of a feed forward network to the current state (or sensor values). The outputs of the neural network correspond to each possible action, and their activation represents the Q-value of that state-action pair. To update the neural network, backpropagation can be used by sending the α from Q-learning as a parameter to backpropagation and setting the target for the output node corresponding to $(1- \gamma) \times r_{t+1} + \gamma \times Q(s_t, a_t)$.

Backpropagation of error is a supervised learning technique popularized by Rumelhart, Hinton & Williams (7). Backpropagation works by assigning blame for error (the difference between the output vector and the desired output vector for a given input vector) to the incoming links to each output neuron. These weights are then adjusted via gradient descent. The error then propagates back to the neurons that caused the incoming links to the outputs to fire. The amount of error assigned is based on the amount of blame assigned to each link and is reduced by a parameter gamma. With the new error assignments, this process can repeat until the input neurons are reached. Backpropagation has been shown to converge and produce robust generalizations of good samples. (7)

NeuroEvolution of Augmenting Topologies

While Q-learning with neural network function approximators can provide excellent performance, doing so requires carefully selecting an appropriate network topology and initial weights. Although in some cases time-consuming manual selection can result in a fairly good representation, more often machine learning algorithms are being used to solve problems in which little is known. In these cases, selecting a reasonable representation can be nearly impossible. Automating this task would therefore be very beneficial. Evolutionary methods have been shown to perform well in automated optimization tasks. (citation?)

NeuroEvolution of Augmenting Topologies (NEAT) is a novel evolutionary method to evolve a controller neural network for reinforcement learning tasks. NEAT uses sophisticated evolutionary mechanisms to evolve not only the network link weights, but also the topology. By evolving the topology of the neural network, NEAT is able to more accurately model the features of its domain without time consuming and inaccurate human tweaking. The three main features that allow NEAT to effectively evolve its topology are “(1) employing a principled method of crossover of different topologies, (2) protecting structural innovation using speciation, and

(3) incrementally growing from minimal structure." (8) Furthermore, NEAT is able to optimize and complexify solutions simultaneously.

Being an evolutionary algorithm, most of the standard procedures apply. There is a population of organisms that perform some task and then are evaluated by their success at that task by some fitness function. This is then used to evaluate which members of the population should breed to produce the next generation. Breeding the best performers provides exploitation of known good behaviors and mutation provides a proper exploration balance of these behaviors. The phenotype of each organism is characterized by its neural network controller – or action selector – that takes as input sensor data of its environment and produces as outputs the action to take.

However, the most compelling part of NEAT as an evolutionary algorithm is the reproductive process, including how the topology is formed. Three features of this process are outlined below. First, to minimize dimensionality, NEAT initializes the population with no hidden layers – just fully connected input to output links – and adds complexity through mutation. Doing so allows only features that improve performance to survive, and thus unnecessary complexity is minimized. The two complexity adding mutations are 'add node' and 'add link'. 'Add node' takes an existing link in the neural network, and splits it in two by adding a new neuron node in the middle, which is then connected to the same neurons the original link was. The other mutation, 'add link' adds a new link between two neurons where there was none before.

Second, flexible genetic encoding is required in order to evolve network structures. In NEAT, each genome is composed of a list of connected genes – which define two node genes being connected, that link's weight, the innovation number, and whether that gene is expressed. This innovation number will allow NEAT to find corresponding genes during crossover. Genes with the same innovation number are lined up, and genes that are not shared by both parents are selected from the fitter parent, or randomly if they are equally fit.

Third, to deal with the fact that topological changes are typically initially harmful until their weights are optimized, NEAT incorporates the concept of speciation. Speciation protects topological innovation by allowing it to optimize – by competing with other individuals in its niche – before competing with the general population. Historical markings are used to divide the population into these niches by comparing the number of different historical markings.

Real Time NeuroEvolving Augmenting Topologies

To provide a more continuous experience in the game domain, real time NeuroEvolving Augmenting Topologies (rtNEAT) will be used, which is a simple modification to NEAT enabling it to work in interactive training scenarios by making the agent replacement mechanism occur in real time. Unlike in NEAT, where

individuals in the population are all replaced in one swoop after every generation, rtNEAT takes individuals out of the environment incrementally, which enables a less jarring experience for a user training a population in real time. After n game ticks, rtNEAT will remove the individual with the worst adjusted fitness. The average fitness of all species is then adjusted. Following this, parents are chosen with appropriate fitness, and a new offspring is created. The new agent then begins its life, joining its associates in the world. (9)

NeuroEvolving Augmenting Topologies + Q-learning

NeuroEvolving Augmenting Topologies + Q-learning (NEAT+Q) is an enhancement to the NEAT algorithm meant to improve learning rate and overall performance. It uses NEAT to evolve a function approximator for the Q function in Q-learning. NEAT+Q has shown good empirical performance – outperforming both NEAT, and Q learning alone in the mountain car domain, and the server-job scheduling domain (10). By automating the search for good representations of a function approximator, NEAT is able to select the topologies and initial weights that enable efficient individual learning. This is a huge improvement over manual selection of neural network topologies, which can be challenging to select and often result in suboptimal representations. In short, NEAT+Q evolves agents that are better able to learn. NEAT+Q is essentially NEAT, which some modifications to how actions are selected at each tick. The Q-learning action-selection policy is implemented, using the evolved network as a function approximator for the Q-function. Also, backpropagation is used to update the neural network function approximator at each time-step, as done in Q-learning with a neural network function approximator (10).

Approach

Evolution has already shown its ability to adapt to changing conditions across many generations. However, this is often too slow, as many domains have hidden changes occurring within an agent's lifetime. Especially in the case of autonomous robots, it would be impractical to have a population of physical robots training and failing (breaking) – one would expect a robot to serve its function without requiring constant replacement. Although memory systems have shown promise in certain adaptive scenarios (3), their ability to adapt appears to be dependent on their ability to model every discrete adaptive condition with a memory system. This suggests they will not exhibit two desirable properties for online adaptive – generalization of adaptation, and ability to model continuous adaptation.

Evolutionary function approximation

To achieve this end, it seems appropriate to use a full online learning system. As the temporal difference method Q-learning has shown good empirical results, we will be exploring its ability. (5) Using a neural network function approximator has shown good empirical results, and improves the ability of Q-learning to generalize

and scale to larger state/action spaces. Using neural network function approximators traditionally required careful manual selection of topology and initial weights, requiring knowledge about the domain that one might not have and expending significant resources to this selection – often causing the selection to be suboptimal. By automating the selection of the network topology and initial weights for the neural network function approximator, these problems can be avoided and solutions to more domains can be achieved in less time. Evolution is an obvious contender for automatic selection, and the NEAT method has already been shown to perform well in this vane. (10) The ability of NEAT to minimize dimensionality should help keep backpropagation effective in updating the Q-function online. Furthermore, evolution with online learning mechanisms has already been shown to improve robustness to unpredictable sources of change (2). Finally, we will substitute NEAT with rtNEAT to make agent replacement occur in real time providing a more continuous experience in the game domain that we will be exploring.

Adaptive Learning Rate

One problem discovered early on, was that Q-learning would over-train its values, slowing its ability to respond to changing conditions, and reaching an unstable state in some cases. Once the optimal action for a given state has a higher predicted long-term reinforcement value than the other actions, further prediction accuracy is not necessary as the current values already result in optimal behavior. Because of this, once the relative values of the Q-function for each state are learned accurately, one can reduce learning significantly with no adverse affects. The advantage of reducing the learning rate at this point is if conditions change, the prior conditions will not be over-trained and it will take less learning to achieve optimal action selection again.

Keeping this idea in mind, having the learning rate adjust dynamically should increase an agent's ability to adapt to changing conditions. When the conditions have recently changed, an agent should have a high learning rate, but when they remain stable, the learning rate should decrease to not over-train. This also simulates annealing, which should solve divergence problems sometimes exhibited by Q-learning with a neural network function approximator. To detect hidden environmental conditions that require change, it seems appropriate to respond to the reinforcement signal. After all, we only need to learn a new strategy if our predictions are off enough that the wrong action is being taken (our currently policy is bad). This tracks changes that require fast learning, and ignores those changes that have no affect on our policy. To accurately model action problems in current strategy, updates to the learning rate should only occur in response to actions based on taking a maximal action.

The following is the update formula for learning rate:

$$LR \leftarrow \alpha \times (1 + (1 - \varsigma) \times \left(\frac{LR}{\alpha} - 1\right) - \varsigma \times P(r)))$$

where LR $\in \mathbb{R}[0, 1]$ is the learning rate to be sent to backpropagation, $\alpha \in \mathbb{R}[0, 1]$ is the maximum learning rate, sent as a parameter to Q-learning, ç is the capriciousness property sent to Q-learning that defines how responsive LR is to recent changes, and r is the reward normalized to $\mathbb{R}[0,1]$, and P is a function modeling the distribution of reinforcement values.

The learning rate formula follows the reinforcement signal, factoring gains from the past. The capriciousness parameter determines the temporal weighted, with a larger value weighted more recent reinforcement values higher.

Experiment setup

Simplified dangerous foraging

The dangerous foraging domain is a fairly simple experiment that acts as a benchmark for adaptive intelligence algorithms. The core idea that makes this domain require adaptation is that there is a hidden property of the world that must be discovered, and then exploited by a modification of behavior. In this case, simply subscribing to a fixed policy would perform badly. However, the level of adaptation (strategy selection) required is small and finite, so we will consider this experiment to provide a baseline for adaptive performance, not a stress test. By keeping the experiment simple, we minimize non-adaptive characteristics of the experiment to see more clear distinctions in algorithm capability to adapt. Since it has been previously used to compare combining adaptive synapses with NEAT and NEAT with recurrency alone, it will be interesting to see how rtNEAT+QwALR compares.

The high level concept of dangerous foraging is fairly simple. An agent will be shown successive random fruits, simply identified by how they look. The agent must choose whether to eat the fruit, or not. Some fruit is good for the agent, while others will make the agent sick. So to perform optimally, the agent must decide to eat the fruit that is good, and not the fruit that will make the agent sick. To make the problem a little less trivial, there will be three different types of fruit seen and three possible actions to take. Each action/fruit pair has a corresponding reinforcement value of 1, 0, or -1. In addition to the fruit sensor, rtNEAT will have a sensor representing the reinforcement signal from the last game tick. Since sensors must activate from (0, 1), 0 will correspond to a -1 reinforcement, 0.5 for 0, and 1 for 1. What makes the task interesting is that during an episode, the reward values of each sensor/action pair will change. The analogy for the changing reward values is that the appearances of fruit changes, changing which fruit is good or bad. One-hundred and eighty fruits were shown before the reinforcement values of each fruit would change. The training was run for sixty episodes.

Q-learning with and without adaptive learning rate, rtNEAT, rtNEAT+Q, and rtNEAT+QwALR were all compared in this domain. Every time an adaptive learning rate was used, the capriciousness parameter was set to 0.9 to maximize the response to changing conditions, since they happen abruptly. The Q-learning parameters were also consistent when used. Γ was set to 0, since action is based on the immediate sensor values, and α was set to 0.5 and the exploration rate for the epsilon-greedy policy for Q-learning was set to 0.1. Whenever backpropagation was used (for Q-learning), 0.5 was used as the discount factor. The population size of rtNEAT was set to 20. Due to the simplicity of the problem, a fully connected input/output network with no hidden nodes was used for testing Q-learning alone.

Robotic simulation dangerous foraging

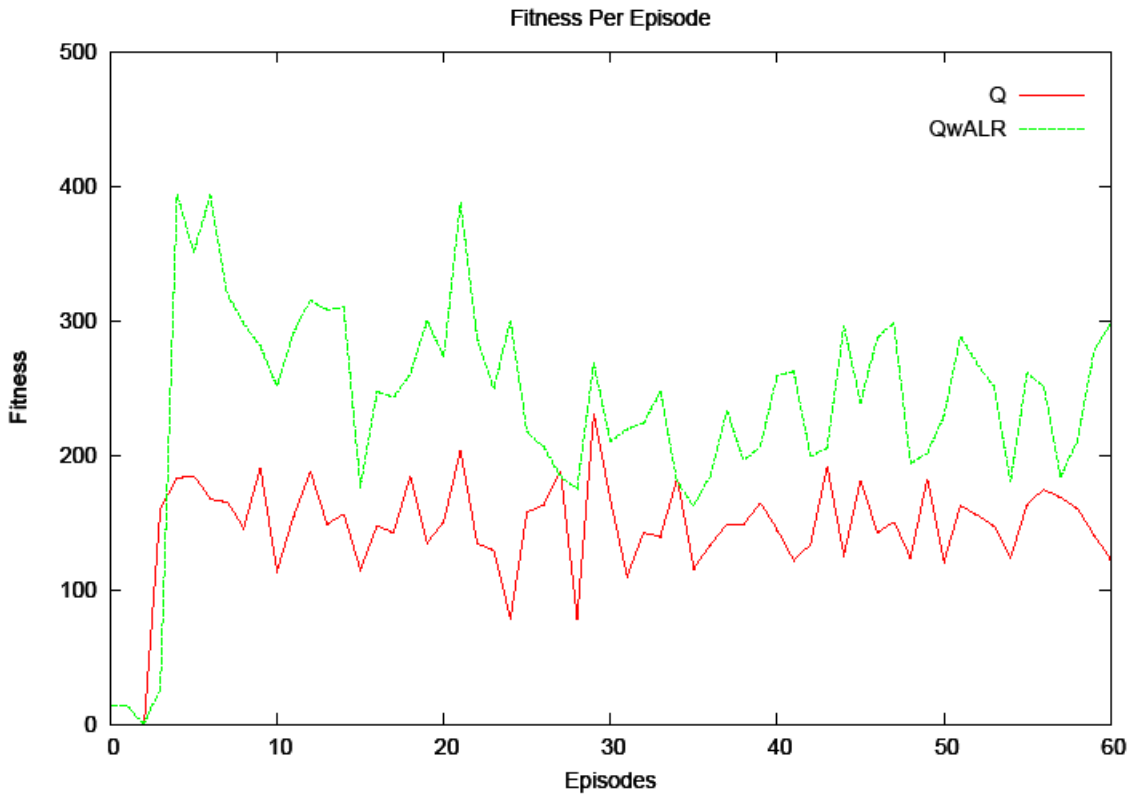
To provide a more compelling real-world scenario, a more complex version of the dangerous foraging experiment was conducted involving a robotic simulation. The game environment takes place in a two-dimensional world. Agents have two wheels that can either be on or off for each round. Agents must explore the environment to find good places to forage. Three different types of "bushes" with "fruit" were placed on the field. Agents begin foraging from a bush simply by being within proximity. To detect the bushes, agents had three radar sensors and a sensor indicated whether the agent was current foraging for each type of bush. Radar sensors indicate the distance to the closest object of that radar sensor's type in relation to the maximum distance of objects on the field. Each radar type was located to detect objects at the same angles: -45 to -5.625, -5.625 to 5.625 and 5.625 to 45 degrees. Agents receive reinforcement from standing in the proximity of a bush. Each bush is assigned either -2, 1, or 2 reinforcement, and these values are changed at 400, and 700 ticks in the episode. An episode ends after 1000 game ticks.

Only rtNEAT and rtNEAT+QwALR were compared in this domain, as the complexity led to QwALR not able to even approach the bushes. The capriciousness parameter was set to 0.5, γ was set to 0.4, and α was set to 0.5 and the exploration rate for the epsilon-greedy policy for Q-learning was set to 0.1. The discount factor for backpropagation was set to 0.5. The population size of rtNEAT was set to 20.

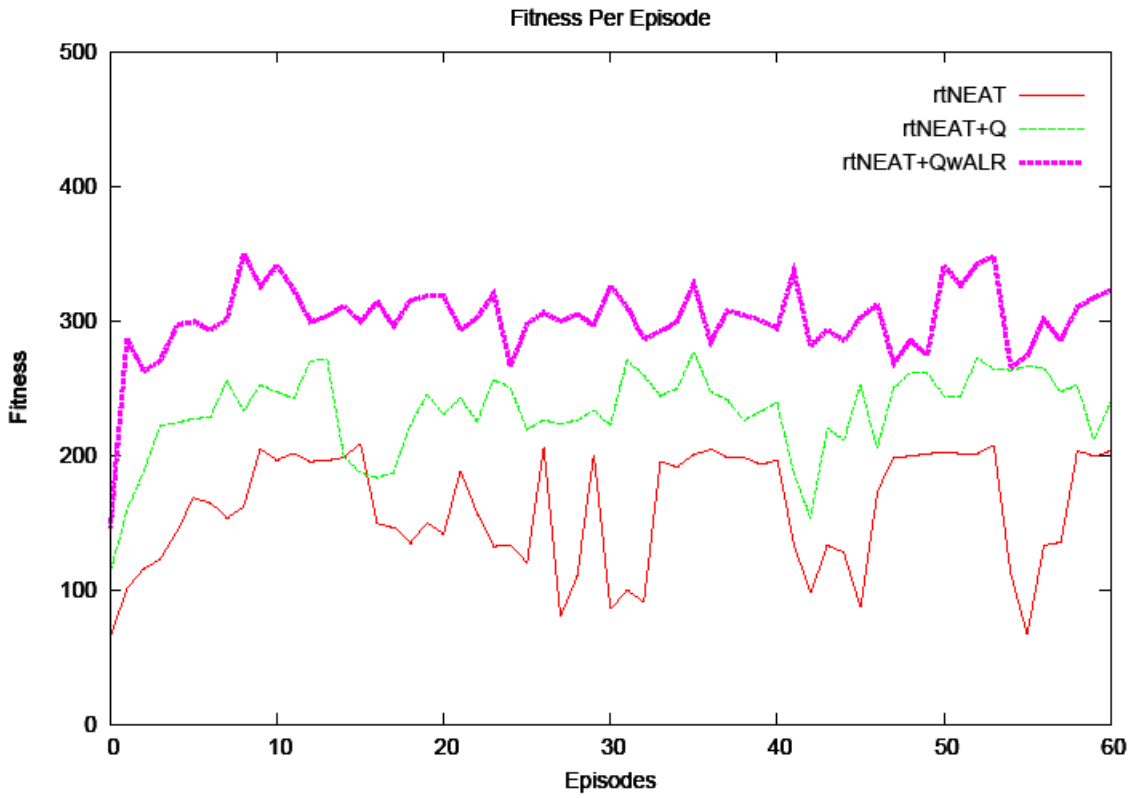
Results

Dangerous foraging

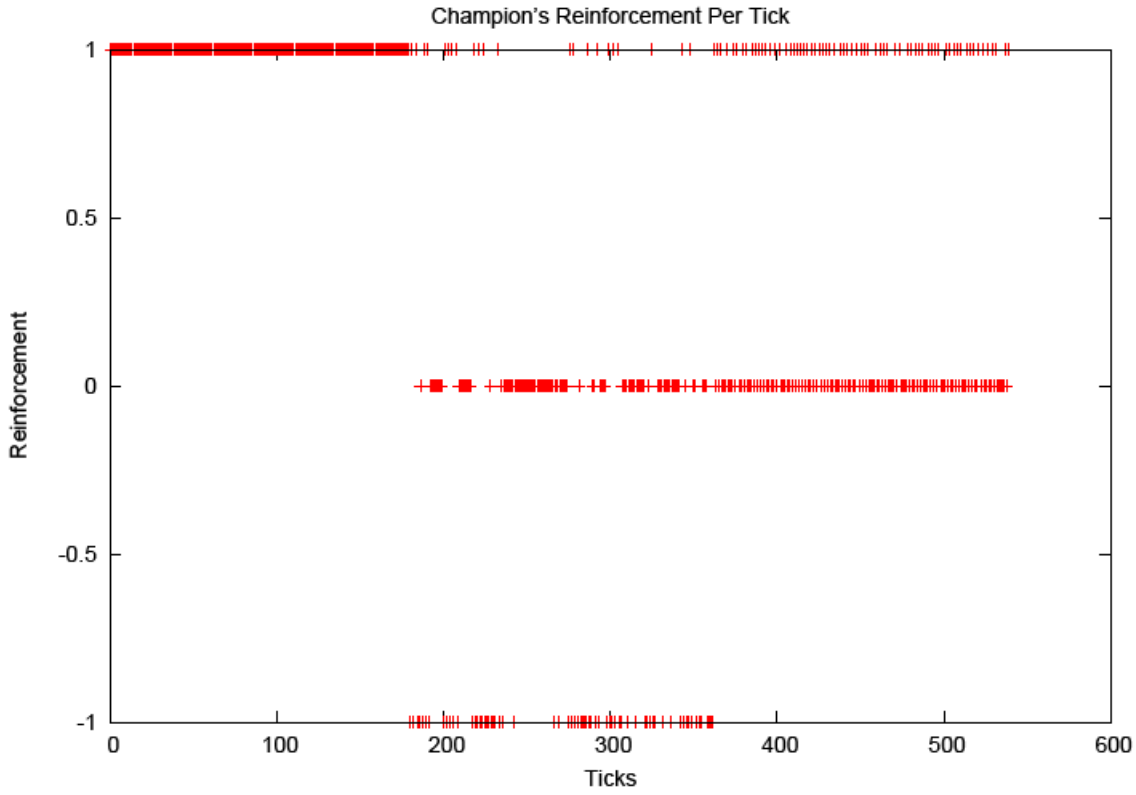
Although this experiment has some similarities to the dangerous foraging experiment used to compare recurrent neural networks with another adaptive synapse algorithm (3), the differences have led to some interesting results.



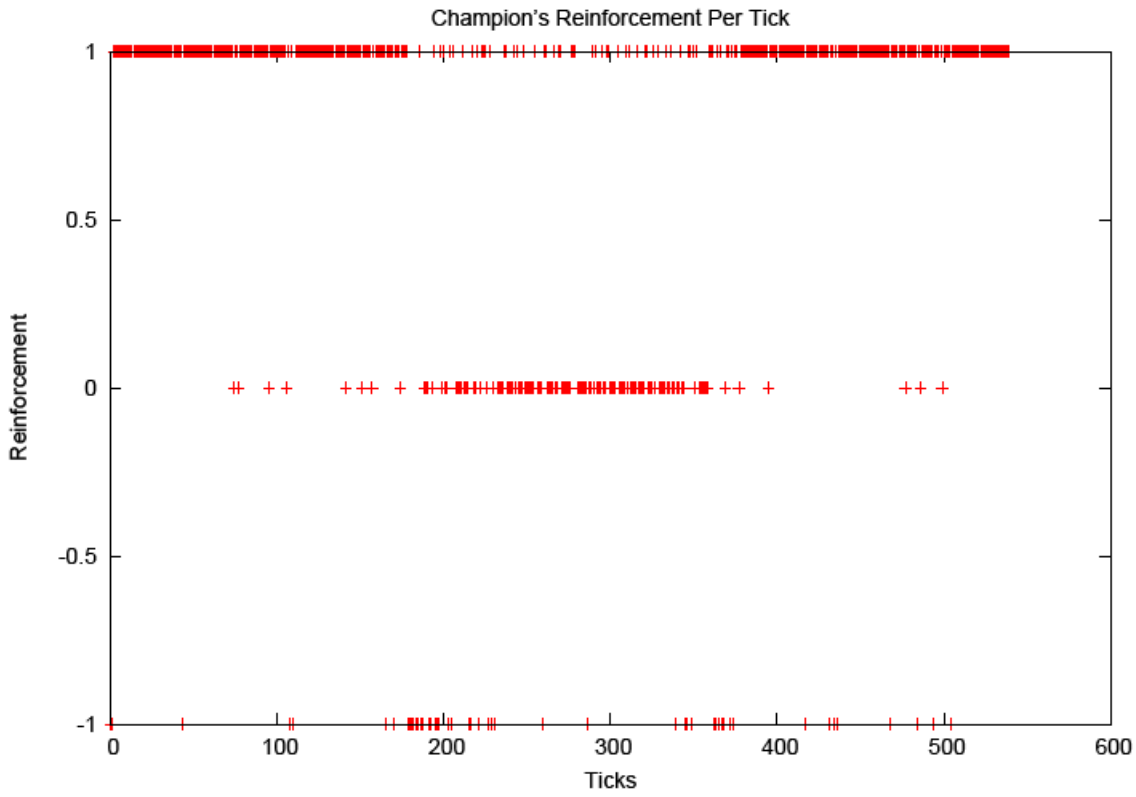
Adaptive learning rate significantly improved performance for Q-learning. The instability of Q-learning is a result of it attempting to converge on a changing target.



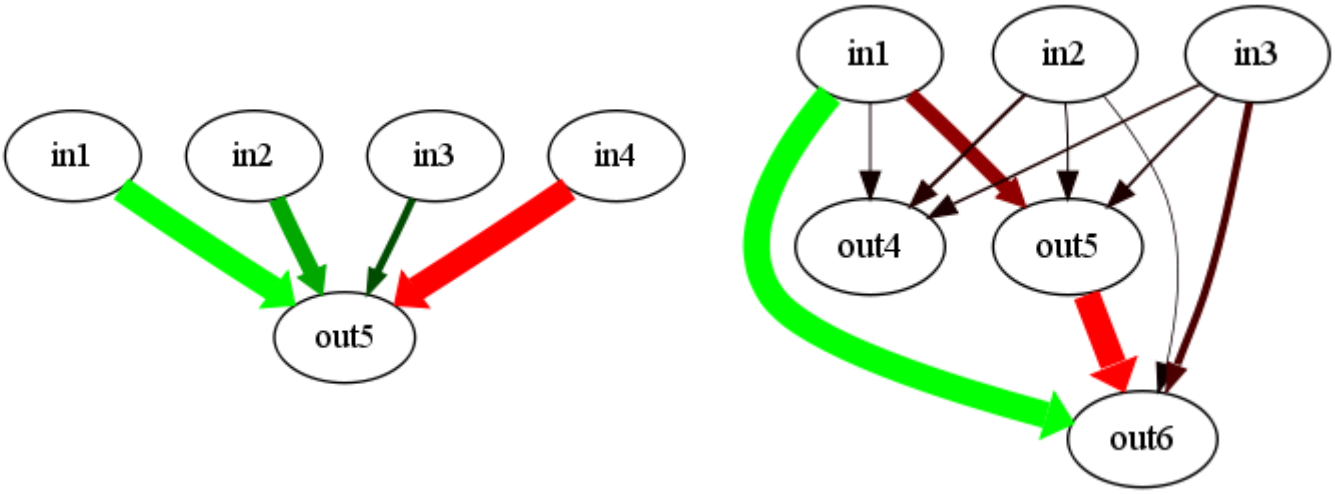
Here we have the evolved agents. rtNEAT alone performs about as well as Q-learning alone. Combining the two gives a good performance boost and adding adaptive learning rate helps a lot – resulting in the best performing algorithm tested. Another interesting point from this graph is that rtNEAT was able to perform reasonably well.



Above we have the reinforcement signal per tick of the champion for rtNEAT at episode 60. Although rtNEAT is not adapting to every change, its fixed policy is able to select optimally for one state, and reasonably well for the others.



Above we have the rtNEAT+QwALR champion at episode sixty. It is clear the agent is adapting to the changing conditions, although its ability to adapt to the second condition is not as solid as the others. However, this is in sharp contrast to the static policy of rtNEAT that is only able to optimally perform under one environmental condition.

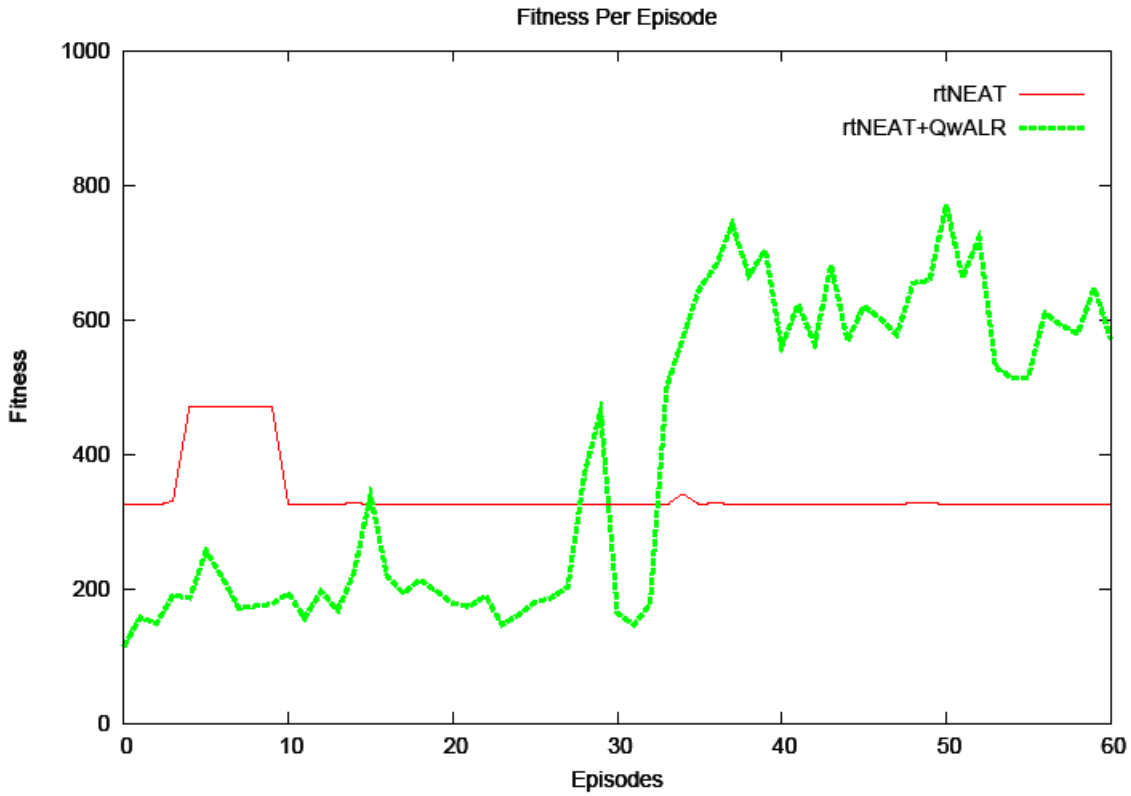


rtNEAT rtNEAT+QwALR

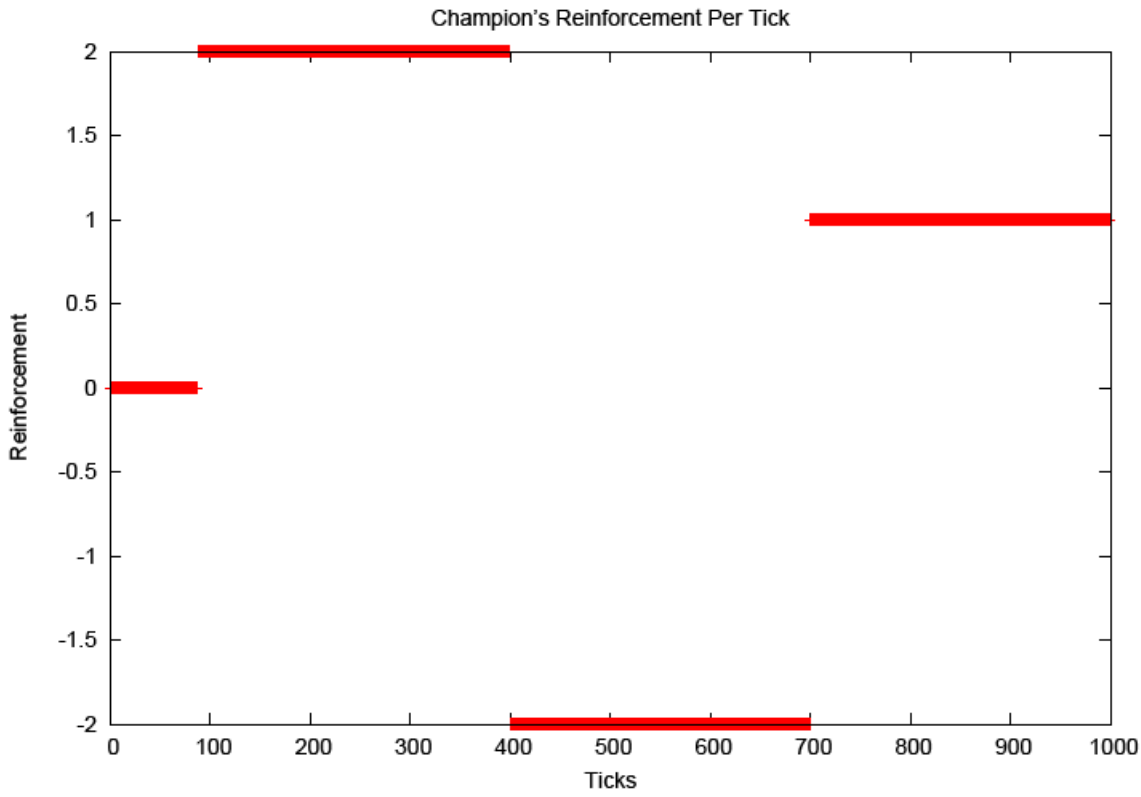
Above we have the champion neural networks from episode sixty of rtNEAT and rtNEAT+QwALR respectively. Their structure is representative of typical performing networks. An interesting result here is that in contrast with previous results comparing evolving recurrent connections to plastic synapses (3), a recurrent connection

was not evolved for rtNEAT, even though the domain was fairly similar. This is most likely due to two things. First, a simpler static solution was easily discovered by evolution that performs reasonably well. Second, there was not enough regularity in this experiment that would have allowed a recurrent solution to significantly outperform the simple static solution. In contrast to the earlier comparison performed by Stanley and Miikkulainen, at each game tick we showed a different random fruit, rather than the same one for a period of time. When the randomness was removed, and fruit was shown in regular intervals, the rtNEAT solution was able to perform better, suggesting its ability to exploit unexpected invariants during training. This would be detrimental to performance where these invariants are not maintained, further supporting the need to generalize adaptive behavior.

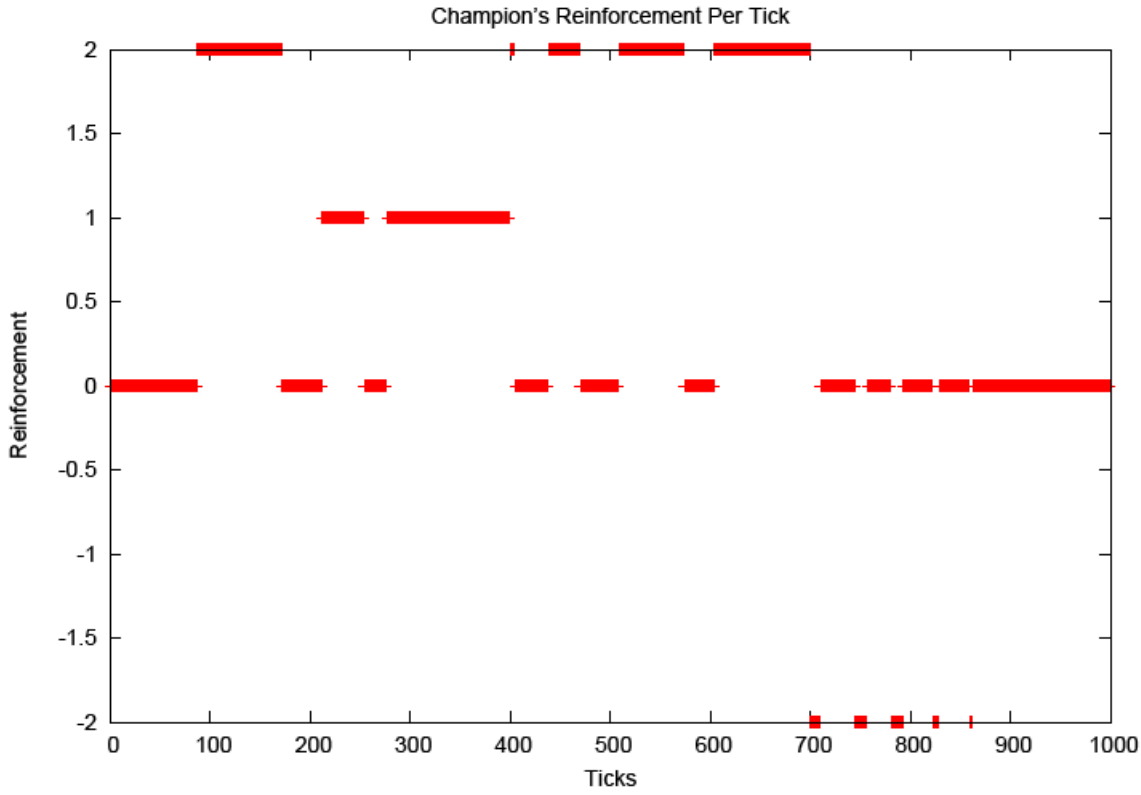
Robotic simulation dangerous foraging



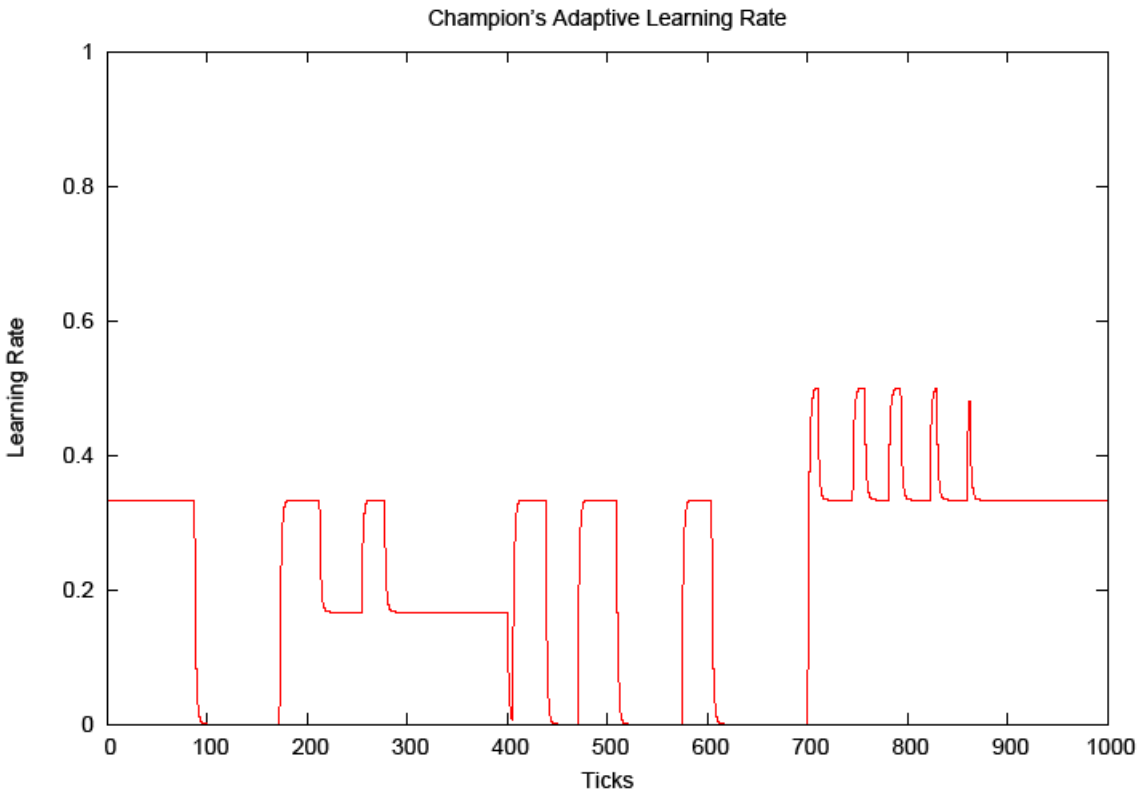
Again in this experiment, rtNEAT finds the optimal static solution. The rtNEAT agents would quickly run to one of the bushes and simply stand there the entire round. rtNEAT+QwALR had a little more trouble learning a good solutions, as its search space was slightly larger due to having to represent every possible action combination in its outputs. However, given enough time, it quickly learned to exhibit adaptive behavior – moving to the next bush after the one it was foraging from starting bearing harmful fruit. Its ability to adapt effectively allowed it to outperform rtNEAT alone.



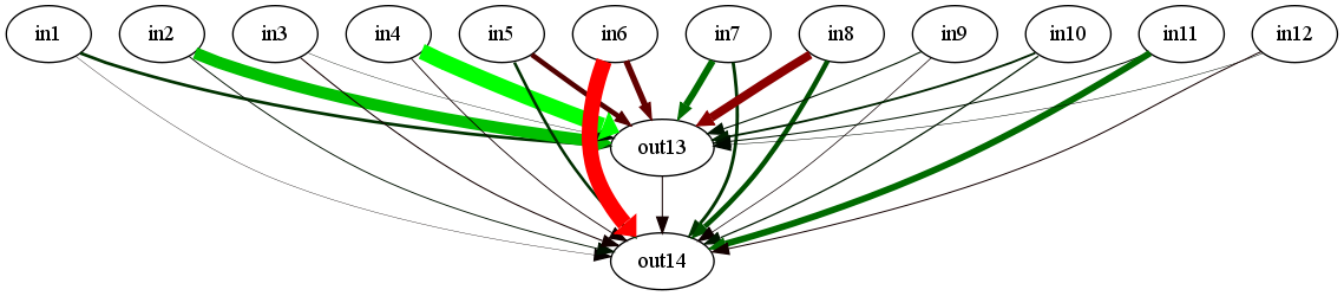
Above is the reinforcement per tick of the rtNEAT champion at episode 60. The 0 reinforcement at the beginning is due to the agent still running toward the first bush. Once the agent arrives, it stays there, and the reinforcement signal assigned to that bush changes twice in its lifetime.



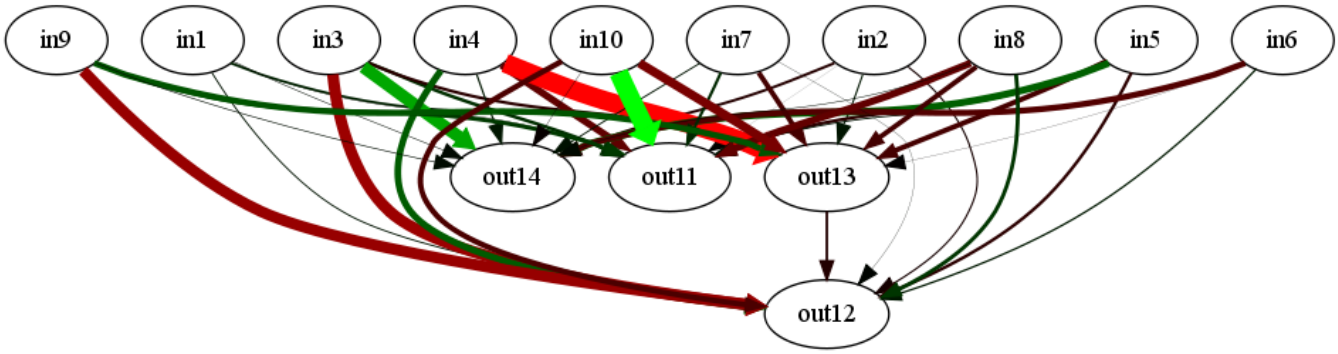
Above is the reinforcement per tick of rtNEAT+QwALR's champion at episode 60. The small missing patches indicate when the epsilon-greedy policy took a random action that moved the agent into a different state. The agent then had to turn around and move back to where it was. Ignoring that noise, this graph indicates the agent ran to the first bush, where it received the highest reinforcement, then it changed bushes to a not quite as good one in anticipation of that bush being assigned the highest reinforcement, which it then waited out. As that bush turned to negative reinforcement, the agent moved away and stopped foraging, occasionally running into the bush due to the random actions taken from the epsilon-greedy policy. This shows rtNEAT+QwALR was able to adapt in this complicated robotic simulation environment, and move between bushes to foraging more optimally than a fixed-policy.



Here we track the learning rate of the champion of rtNEAT+QwALR at episode sixty. The adaptive learning rate quickly responds changing conditions that are indicative of a need for policy change. Although there is the occasional spike due to epsilon-greedy action, the algorithm still performs fairly well, resulting in the increased performance compared to rtNEAT alone.



rtNEAT



rtNEAT+QwALR

The champion neural networks at episode sixty are shown above.

Conclusion

There are often unknown hidden variations in an environment that change during an agent's lifetime. In these cases, to perform optimally, an agent must adapt their behavior online. We have explored one possible solution to the problem – combining evolution with an online learning mechanism. After enhancing NEAT+Q to more quickly adapt to changing conditions (using an adaptive learning rate), we tested its performance against its base components. Using evolution with Q-learning was shown to improve performance over using evolution and Q-learning alone. Furthermore, adaptive learning rate greatly improved performance when adaptation was necessary. The resulting algorithm rtNEAT+QwALR exhibited the best performance in the dangerous foraging domain because of its ability to adapt its behavior when hidden environmental conditions change.

Future work

Reinforcement signal probability distribution modeling

One limitation of this function is the need to select a good probability distribution function for reinforcement values (P), as a bad model may lead to undesirable learning rate adjustments. However, in practice this is significantly easier than attempting to model adaptive behavior directly. If the type of distribution is known, it would be fairly easy to add a formula to calculate the parameters of that distribution (such as μ and σ for the normal distribution). If the type of distribution is not known, a new neural network function approximator might be used to attempt to model this function.

Model partially observable Markov decision process (POMDP) more accurately

Although modeling robotic agents as MDPs is a decent approximation, it would be compelling to see if more accurately modeling their work as a partially observable Markov decision process (POMDP) and see what kind of performance increases can be gained from this. One potential avenue for exploration would be to replaced Q-learning with the BEL-RTDP algorithm which models the policy as belief state values (11).

Use sample efficiency

If agents require many episodes of exploration in their environment to effectively adapt and explore their learning potential, this can be costly to the user training those agents. If the agents are robots, they must run during this time. If the agents are simulated in a video game, the user must watch them explore for many episodes before they evolve. By exploiting the off-policy nature of Q-learning, sample efficient NEAT+Q allows agents to train additional episodes offline, trading cpu-resources for training time (12). In the case of video games, this could actually reduce cpu-usage overall since repetitive sensor data gathering is often more of a bottleneck than Q-learning.

Bibliography

1. *Evolutionary Robots with On-line Self-Organization and Behavior Fitness.* **Floreano, Dario and Urzelai, Joseba.** 2000, Neural Networks, pp. 13:431-434.

2. *Levels of dynamics and adaptive.* **Blynel, Jesper and Floreano, Dario.** 2002. Proceedings of the Seventh International Conference on Simulation of Adaptive Behavior.

3. *Evolving Adaptive Neural Networks with and without Adaptive Synapses.* **Stanley, Kenneth O., Bryant, Bobby D. and Miikkulainen, Risto.** 2003. Proceedings of the 2003 IEEE Congress on Evolutionary Computation.

4. *Elevator group control using multiple reinforcement learning agents.* **Crites, Robert H. and Barto, Andrew G.** 1998, Machine Learning, pp. 33(2-3):235–262.

5. **Watkins, C.** Learning from Delayed Rewards. s.l. : PhD thesis, King's College, Cambridge, 1989.

6. **Sutton, Richard S. and Barto, Andrew G.** Reinforcement Learning: An Introduction. Cambridge, MA : MIT Press, 1998.

7. *Learning internal representations by error propagation.* **Rumelhart, David E., Hinton, Geoffrey E. and Williams, Ronald J.** 1986, Parallel Distributed Processing, pp. 318–362.

8. *Evolving neural networks through augmenting topologies.* **Stanley, Kenneth O. and Miikkulainen, Risto.** 2002, Evolutionary Computation, pp. 10(2): 99-127.

9. *Evolving neural network agents in the NERO video game.* **Stanley, Kenneth O., Bryant, Bobby D. and Miikkulainen, Risto.** 2005. Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games.

10. *Evolutionary Function Approximation for Reinforcement Learning.* **Whiteson, Shimon and Stone, Peter.** 2006, Journal of Machine Learning Research.

11. *Solving large POMDPs using real time dynamic programming.* **Bonet, B. and H.Gefner.** 1998. AAAI Fall Symposium on POMDPs.

12. *Sample-Efficient Evolutionary Function Approximation for Reinforcement Learning.* **Whiteson, Shimon and Stone, Peter.** July 2006. Proceedings of the Twenty-First National Conference on Artificial Intelligence. pp. 518–23.