DOI: http://dx.doi.org/10.7551/978-0-262-31050-5-ch048

On the Emergent Behaviors of a Robot Controlled by a Real-Time Evolving Neural Network

Walter O. Krawec[1]

[1]Stevens Institute of Technology, Hoboken, NJ 07030
wkrawec@stevens.edu

Abstract

In this paper we apply a real-time evolving neural network which uses a hill-climbing algorithm capable of adapting not only a network's synaptic weights but also its topology (creating a recurrent neural network). We then apply this network to a robot in a simulated environment. By equipping the robot with a minimal set of instincts and a short-term memory system (to facilitate reinforcement learning), we observe that several strategies developed which pass the emergent behavior test of (Ronald et al., 1999). In particular, we see robots learning behaviors that are not rewarded by the environment.

Of course a hill-climbing algorithm is more likely than a genetic-algorithm to get stuck at a local optimum, we argue that, despite this, the method described here has several unique advantages. In particular, it allows us to create a single persistent robot that slowly learns and "grows up" as described in (Ross et al., 2003). With our system, it is an individual that learns not a population of individuals, and our learning is continual (e.g. there is no need to reset the robot to some starting position to evaluate the fitness of a particular network).

We conclude with several future problems and applications. For instance, we describe a simple mechanism allowing a network to be copied to embedded hardware whenever a network connection is available to a PC (which is responsible for the memory and time intensive task of evolving the network). This mechanism does not require a continual link to a PC. We also discuss the possibility of creating a distributed evolving neural network system.

Introduction

In this paper, we apply a real time evolving neural network (ENN) to a (currently simulated) robot which begins its existence without any prior knowledge of itself or its environment. That is, the robot has absolutely no idea as to what its various inputs mean, or what its outputs do. By equipping the robot with a simple short term memory, along with a variety of very basic "instincts" and "reflexes", we allow the robot's neural network to evolve itself (in realtime). This evolution adjusts not only synaptic weights, but also the network's topology (creating a fully recurrent network).

To accomplish this, a hill-climbing approach is used instead of the more typical genetic algorithm (see (Junfei et al., 2007), (Floreano & Keller, 2010), (Stanley & Miikkulainen, 2002), (Cliff et al., 1992), and (Stanley et al., 2005) for some examples). By using a hill-climbing approach, we believe we achieve more interesting personalities (term used informally of course; the point is, that by using one network and slowly adapting it, certain traits unique to a particular robot are preserved throughout its lifetime - traits that may be lost if a multi-generational GA were used); furthermore, it allows us to achieve certain items mentioned in (Ross et al., 2003) listing the conditions required for allowing a robot to "grow up". Furthermore, our algorithm does not require multiple agents to be evaluated and then reset (or even a single agent to be run, evaluated, then reset); instead a single robot may operate while continually evolving as it gains new experiences.

Evolution is facilitated by equipping each robot with a short-term memory (STM). This memory stores recent actions along with the rewards/penalties given for their performance. These rewards/penalties are provided by a robot's minimal instincts. Furthermore, STM is not a look-up table. Indeed it may be that some entries are incorrect and others are never filled. Furthermore, STM only holds data for a (relatively) short time; it is up to the ENN to remember the data and avoid situations that result in penalties (such as crashing into walls).

Our simulations will consist of one or more robots equipped with a variety of sensors (e.g. proximity, light, sound etc.). We use a very minimalist reward system as driven by simple instincts. Specifically we use three instincts: Pain, Boredom, and Human Response (allowing a human operator to train a robot to perform a specific task if so desired). By implementing these few rewards, our robot's neural network is able to quickly learn to avoid walls. However other very interesting behaviors may emerge including:

1. Closely following a wall (despite binary proximity sensors).
2. Learning to follow a second robot which has learned to avoid walls
3. Some robots learn to group together while others avoid

each other

4. Learn to seek, avoid, or ignore light

We use the emergent behavior test defined by (Ronald et al., 1999) for our purposes and show that many unexpected strategies developed by our robots pass this test. Informally, however, the behaviors developed are considered to be emergent due to the fact that such strategies are not directly rewarded (and hence not expected). For example, there is no reward for grouping together, yet many robots developed behaviors that, in addition to avoiding obstacles, sought out other robots in the simulation. We will describe this in more detail later.

We also demonstrate other advantages to our real time ENN approach. This includes the ability of allowing a robot to adjust to new I/O in real time (where the new I/O may be added "on-the-fly" while the robot is in operation). We make use of this ability to slowly train a robot to perform more complicated tasks from simpler ones (another condition of (Ross et al., 2003)). Also our ENN very easily evolves to memorize certain patterns in input data which may be beneficial to future work.

Real Time Evolving Neural Network

We now describe briefly the evolution algorithm used in our experiments. Given a neural network $\mathcal{N}$, we assume that there exists a fitness function f mapping $\mathcal{N}$ to $f(\mathcal{N}) \in \mathbb{R}$ such that $f(\mathcal{N}_1) > f(\mathcal{N}_2)$ implies that $\mathcal{N}_1$ is a "better fit" to some data set than $\mathcal{N}_2$. We formalize this later by constructing such a fitness function based on a robot's short term memory. In this section however we simply describe the evolution of a neural network with respect to this function f.

Our algorithm begins with a simple feed-forward network consisting of a single neuron for each input and output. Furthermore, input neurons are assigned linear activation functions while output neurons (and indeed any other hidden node) is assigned a sigmoid function (specifically $\frac{1}{1+e^{-x}}$). Also, if requested, our network may begin with a collection of random hidden neurons. These neurons are connected randomly to the network.

A single iteration of the evolution algorithm will take as input a network $\mathcal{N}$ along with fitness function f and output a network $\mathcal{N}'$ (possibly the same network) such that $f(\mathcal{N}') \geq f(\mathcal{N})$. This is achieved by taking N copies of $\mathcal{N}$ and modifying each (independently) according to the following rules:

1. If a neuron or synapse was added or removed within the last T evolution cycles, we modify 15% of the synaptic weights. This is to allow changes to the network's topology to "settle" optimally.

2. With probability p_1, we add a new random neuron

3. With probability p_1 we remove a random neuron (but not I/O neurons)

4. With probability p_2 we add a random synapse connecting two neurons (chosen at random; possibly the same neuron creating a loop)

5. With probability p_2 we remove a random synapse

6. With probability $1 - 2p_1 - 2p_2$, we modify 15% of the synaptic weights.

From these $N + 1$ networks (for we consider the original $\mathcal{N}$), we choose $\frac{N+1}{K}$ of the very best (those with the highest value according to f) and $\frac{N+1}{2K}$ random networks to which we apply a second iteration of the evolution algorithm to each of these separately (and indeed, this recursion repeats a total of R times). Note that if a neuron/synapse was added or removed in any iteration, further applications of the evolution algorithm will simply adjust synaptic weights. Finally, we choose the very best of the resulting $N+1$ networks and output it. The original network $\mathcal{N}$ is replaced by this new network.

For our experiments, we found good results by setting $p_1 = 0.004$, $p_2 = 0.006$, $N = 100$, $R = 1$, and $K = 10$. Of course larger values of N, K, and R should lead to faster learning though it does slow the algorithm.

Furthermore, every change made to a network is logged in a simple evolution tape. By following this tape, we may separately re-create the evolution of a network. This permits us to create a multi-threaded application where one thread is devoted to running the evolution algorithm while another thread keeps an independent copy for use in real-time. Every few cycles, the evolve tape may be requested (rather, only the latest changes made to the evolve tape). From this tape, the networks may be synchronized.

Additionally this evolve tape allows us to easily run an evolving neural network on embedded hardware. This is accomplished by using a PC to evolve a network (which requires substantial time and memory) while the embedded device simply requests the latest evolve tape every so often. From this tape, the embedded device may very easily build a local copy of the network to run at will. This (and other applications of the evolve tape) are described in a later section.

Short-Term Memory

In the previous section we described the evolution algorithm with respect to a fitness function f. We now describe how this function is actually constructed.

Every robot is equipped with a form of short-term memory (STM). This memory is responsible for holding a limited amount of "possibly useful" information that a neural network should attempt to capture. We say "possibly" useful since it may be that a particular action was mistakenly

added to STM. Hence our mechanism must be able to guide the evolution of a neural network but not maintain its contents for too long a time. To achieve this, we propose two methods: U-*Learning* and *EL-Learning*. We will describe their application and also mention some of the advantages and disadvantages to each.

We begin with U-Learning (the U stands for *utility*). This method is similar to Q-Learning (Watkins & Dayan, 1992) in that we will store a matrix mapping state/action pairs to rewards. However the method differs in that U-Learning's goal is only to exist in the short-term whereas Q-Learning's goal is to fully explore the reward space.

We begin with a zero matrix $U^0 \in \mathbb{R}^{2^m \times 2^n}$ where m is the number of inputs to our network and n is the number of outputs. The superscript is used to index the time at which this U matrix is valid. Given utility matrix $U^t = (u^t_{i,j})$, the value $u^t_{i,j}$ represents the reward (or utility) for applying action j from state i. States and actions are computed in the obvious way: given input vector $(x_0, x_1, \cdots, x_{m-1}) \in \{0,1\}^m$ and output vector $(y_0, y_1, \cdots, y_{n-1}) \in \{0,1\}^n$, then the state (respectively action) is simply $\sum x_i 2^i$ (respectively $\sum y_i 2^i$).

When in operation, if a robot receives a reward or penalty $r \in \mathbb{R}$ (exactly how a robot receives these rewards/penalties is described later) for performing a certain action J given state I, then we construct a new matrix $U^{t+1} = (u^{t+1}_{i,j})$ where:

$$u^{t+1}_{i,j} = \begin{cases} u^t_{i,j} + r & \text{if } i = I \text{ and } j = J \\ u^t_{i,j} & \text{otherwise} \end{cases}$$

Finally, since we are only interested in storing data in the short-term, every T cycles, we construct the matrix: $U^{t+1} = \eta U^t$ where $\eta \in (0,1)$. For our experiments, we set $T = 10$ and $\eta = 0.8$.

From all this, we may construct our fitness function with respect to U^t. This function, which takes as input a neural network $\mathcal{N}$ (with m inputs and n outputs) and U-learning matrix U^t is defined in pseudocode as:

function UFitness($\mathcal{N}, U^t$)
$sum = 0$
for $j = 1$ to M **do**
  Choose π_j a random permutation of $\{0, \cdots, 2^m - 1\}$
  for $i = 0$ to $2^m - 1$ **do**
    Run $\mathcal{N}$ on input representing state $\pi_j(i)$
    for $k = 0$ to $2^n - 1$ **do**
      $p = u^t(\pi_j(i), k)$ ▷ $u^t(x,y) = u^t_{x,y}$
      for $l = 0$ to $n - 1$ **do**
        if l'th bit of k is 1 **then**
          $p = p \times (l$'th bit of output of $\mathcal{N})$
        else
          $p = p \times (1 - l$'th bit of output of $\mathcal{N})$
        end if
      end for
      sum = sum + p
    end for
  end for
end for
return (sum/M)

Note that we must randomly permute the states else a network evolves to expect inputs in order 0, 1, etc. We average the fitness over M distinct permutation. M of course should depend on the state size of a network, for our experiments we found good results with $M = 10$.

We now present a second method of handling a robot's STM which we call EL-Learning (the EL stands for *Evolution List*). This method begins with an empty list $E^0 = \emptyset$ (again the superscript determines the time index at which this list is valid). Elements in E^t will consist of 4-tuples of the form $(\mathbf{x}, \mathbf{y}, r, T)$ where $\mathbf{x} \in \mathbb{R}^m$ (where m is the number of inputs to our ENN), $\mathbf{y} \in [0,1]^n$ (n being the number of outputs), $r \in \mathbb{R}$ is the reward for outputting $\mathbf{y}$ given input $\mathbf{x}$, and T is the lifetime of this element (measured in evolution cycles; if $T = \infty$ its lifetime is infinite).

Whenever a reward is received, we construct a new list $E^{t+1} = E^t \cup \{(\mathbf{x}, \mathbf{y}, r, T)\}$ where $\mathbf{x}$ is the current input vector, $\mathbf{y}$ the current output vector, r the reward value (possibly negative implying a penalty) and T is this element's lifetime (depending on the type of reward, in our experiments this value ranges from $50 - 200$). Furthermore, if $|E^{t+1}| \geq M$ for some M we remove the element from E^{t+1} with the smallest value of T. This is done not only to keep our list at a manageable size, but also to keep its purpose as a short-term memory mechanism - not a look-up database. It is possible to set $M = \infty$ which implies the list will continue to grow with elements removed only if $T = 0$.

Finally the fitness function with respect to E^t is defined as:

$$f_{E^t}(\mathcal{N}) = \sum_{e \in E^t} e.r(n - \delta(\mathcal{N}(e.\mathbf{x}) - e.\mathbf{y})), \qquad (1)$$

where $\delta(x, y)$ is the usual Euclidean distance squared.

After running our experiments multiple times, we saw that the resulting ENNs evolved using U-Learning or EL-Learning behaved differently. Due to the nature of the evolution process, ENN's not only evolve to achieve a higher fitness function, but they also tend to "remember" the order the STM data is sent to it. Because of this, EL-Learning will typically create a network that learns to expect its input in the order presented in the list and will perform different actions given a different input order (though we may undo this by choosing a random permutation as with the U-Learning fitness function). Since with U-Learning, this order is randomized (and indeed this is the reason for choosing a random permutation; else a network learns to expect its inputs in order $0, 1, \ldots$), the resulting network is usually more "stable". However EL-Learning does tend to produce more interesting behaviors.

Furthermore, EL-Learning permits us to easily use analog inputs whereas U-Learning is binary by nature. We tested this by developing a simple game where an ENN is in charge of shooting at an enemy ship (this enemy ship is floating in space and controlled by a human operator). The inputs to the network are the enemy ship's x and y velocities and its x and y coordinates. The output of the ENN is a value between 0 and π which is translated to the gun's rotation. After watching a human point the gun for a short time (less than a minute with less than 100 rounds), the ENN is able to very accurately point the weapon (usually after running the evolution algorithm for only 20-100 iterations). This demonstrates an ENN's ability to learn a continuous space with only minimal training.

Finally, U learning requires maintaining in memory a rather large matrix (though since most of the elements are zero as we show later, memory may be saved by simulating it as a list) however the evolution tends to be faster and more stable. With further work, we believe that EL-Learning should not only be able to reliably create a stable ENN but also allow for more interesting behaviors to emerge.

Reflexes, Instincts, and Decision Paths

We now answer the question as to how data is inserted into STM. Each robot is equipped with a very minimal set of reflexes and instincts. Reflexes, when triggered (either by the environment, instincts, or by the ENN itself), take full control of the robot for a short amount of time before returning control back to the ENN. A reflex may only control a robot's outputs and may be used to avoid dangerous situations (such as turning away from a wall we just crashed into), or to help the ENN with complicated motion tasks (e.g. moving a leg forward on a legged robot).

Instincts are also minimal subroutines however they may only be triggered by the environment and they do not directly control a robot (though they may trigger a reflex). When triggered, an instinct will provide a reward or penalty to the robot's STM. It is from this data that the robot uses to evolve. Examples of instincts include the before mentioned pain, boredom (to prevent a robot from performing the same action for too long), and human input (to allow a robot to be taught from a human).

Of course instincts are useful to provide rewards or penalties for certain instantaneous actions (e.g. penalizing the action of moving straight when a forward pointing proximity sensor reports an obstacle); however there might be several "decisions" made by an ENN before an instinct was triggered that led to this reward or punishment. Also, we should provide some small reward for actions that do not lead to an instinct being triggered. To this end, we introduce "decision paths" which is a secondary form of STM. Essentially, this mechanism will log a robot's actions in some time interval $[t_i, t_{i+1}]$ (where $t_0 = 0$, and $t_{i+1} = t_i + T$ for some $T > 0$). This log is a list of state/action pairs (s_i, a_i) where a_i was the action taken by the ENN at state s_i (note that if EL-Learning is used, s_i and a_i are vectors). An element of this form is added to the list at fixed intervals $T' < T$ and potentially also whenever a state or action changes. A decision path is therefore an ordered list $P = \{(s_i, a_i)\}$ where the state action pair (s_i, a_i) occurred in time before (s_j, a_j) for all $i < j$.

When T units of time have expired, we take our decision path $P = \{(s_i, a_i)\}$ and add to primary STM (either a U-Learning matrix or an EL-Learning list) the triple (s_i, a_i, r_i) where $r_i = \eta^{|P|-i}$ with $\eta \in (0, 1)$. That is, since we've avoided triggering an instinct we classify the last actions as good however at a discounted factor the further in the past they occurred.

If, however, an instinct is triggered with reward value $w \in \mathbb{R}$ before T units of time have elapsed, we then take our decision path P and add to primary STM the triple (s_i, a_i, r_i) where $r_i = w\eta^{|P|-i}$ with $\eta \in (0, 1)$. That is, our discount factor now depends on the instinct's reward value.

Hence, instincts that provide a penalty will result in any action leading up to it to be considered *potentially* incorrect (the further in the past, the less this is penalized) and likewise any instinct that provides a positive reward will result in actions leading to it to be potentially correct. Since STM slowly loses data over time, any incorrect rewards or penalties added will eventually be discarded or replaced by newer data as the robot discovers it.

Evolving in Stages; or The Persistence of Long-Term Memory

One concern the reader may have is that our ENN (which may be considered the robot's Long-Term Memory) will lose its memory once the STM does. That is, if given two fitness functions f_A, f_B where B is a proper subset of A, is it the case that $f_A(\text{evolve}(\mathcal{N}, B)) \ll f_A(\mathcal{N})$ assuming $\mathcal{N}$ has been evolved over time using A; i.e. does evolving a network with respect to B (which has strictly less data than A) cause our network to completely disregard prior information learned from A?

While we are still investigating this, it doesn't seem to matter very much that the STM only retains partial reward information. Over time, as the STM loses its memory and the ENN continues to evolve over the degraded STM set, the network may "forget" certain actions. However it seems to quickly recover its memory when presented with the proper reward. Furthermore, due to the decision path system, if a robot doesn't use a particular sensor (or subset of sensors) for some time, the ENN may forget what action to take when those sensors are used again. None the less, it seems to be the case that the robot can quickly recover that knowledge. Besides, this is a problem that other living organisms face (e.g. humans).

Finally, we point out that in operation the STM only holds a small fraction of reward information yet this allows the

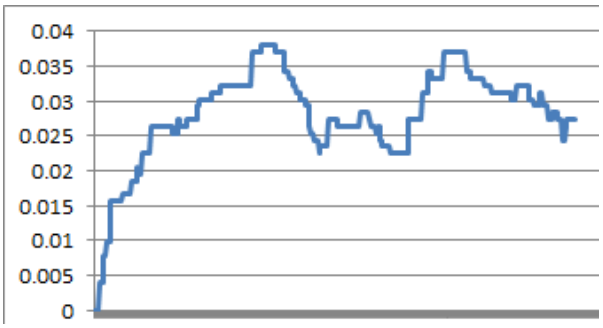

Figure 1: Graph of a robot's STM contents over time (x-axis). The y axis plots the proportion of the U-matrix entries that are non-zero. We notice that at the start of the simulation, the contents of STM quickly grows (as the robot crashes into obstacles for example), peaks, then dissipates (as the robot settles on a strategy/behavior). The memory peaks again when new information is received (an instinct being triggered for example due to an unexpected circumstance or loss of long-term memory). Note however that the content in these simulations doesn't exceed 4 percent of all possible state/action pairs. This is in line with the goal of STM - it is not to serve as a general look-up table for every possible action but meant only to guide an ENN's evolution.

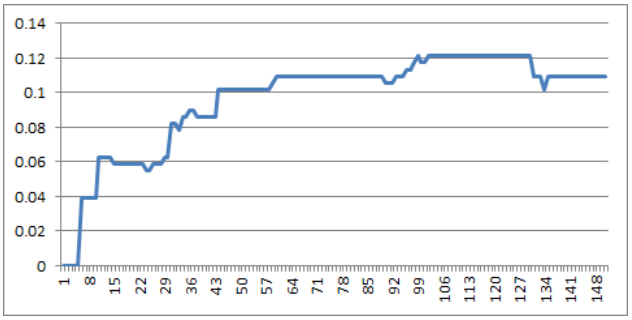

Figure 2: Similar to figure 1 except using a robot with a smaller input set. Here we note that again the STM contents quickly fill then stabilize around 11 percent. The increase around time 90 is when we attempt to teach a robot a new task. Once this task is learned, the STM once again decreases (around time 135) to 11 percent.

robot to perform very well in an environment. See Figures 1 and 2 for a graph of the STM's contents over time (using U-Learning).

We also note that we are able to develop a robot in stages. For example, we may permit a robot to run on its own for some time learning to use its proximity sensors. Then, after this has been learned, we may place it in a new environment and/or teach it to use a different subset of sensors. Left on its own, a robot can usually quickly learn not to crash into walls however it may ignore its other sensors. However we may, at any time, "teach" it to perform certain actions with its other sensors. How exactly this teaching is accomplished is described later (we permit the human operator to use only minimal signals/hints). Also, how this affects STM is shown in Figure 2.

Advantages to using a Hill-Climbing Approach

In (Ross et al., 2003), the authors described 5 conditions allowing a robot to "grow up". They are:

1. It is the individual that develops rather than a system of individuals

2. Involves acquiring a hierarchy of skills where the acquisition of new skills is facilitated by already acquired ones.

3. Not necessary to learn one skill at a time; learning is continual

4. The reason why one action is preferred over another changes with experience

5. In the process of development, the individual becomes capable of purposeful action on longer time spans.

Of course there is the obvious problem that a hill-climbing (HC) approach may more likely settle at a local optimum (though many humans also do this - we call it being "stuck in a rut"). However by having a network slowly grow by only altering its structure slightly at each iteration (as opposed to a genetic algorithm (GA) where several different competing structures are considered and at any time, a network may be replaced by one that is radically different) we satisfy condition 1.

Furthermore, because a network slowly grows in this fashion, certain behaviors in a robot may develop and present themselves, then later lie dormant only to reappear much later in the robot's life. This happened in our simulations multiple times. For one particular example, we had a robot that learned to avoid walls but would, at regular intervals, perform a zig-zag motion (note that there was no reward for this particular action; it was just a developed personality unique to this particular robot). This behavior eventually disappeared as the robot continued to evolve (satisfying condition 4). However, much to our surprise, this zig-zag motion reappeared much later in the simulation. Whatever neural structure that created this behavior remained and was able to re-emerge later; something that would be unlikely in a GA where we might have thrown away this particular population member. A HC algorithm however maintains much of a robot's past. We return to the other conditions of growing up later.

There may be some interesting future work combining the genetic algorithm and hill-climbing approach. Indeed, we may begin by using a GA to evolve a decent population;

each robot may then use one of the networks. Then by using a HC approach as described above, we permit the robot to continually learn and grow.

Experiments

We tested our design in a simple simulated environment. A robot is controlled by a single ENN which communicates with the simulator over a standard network connection (hence the simulator may run on a different computer). If an experiment calls for multiple robots running within the same simulated environment, each of these robots has its own ENN and STM. In fact, each robot runs as its own process and the only communication between these robot processes is through whatever indirect means is available in the simulator (e.g. crashing into one another).

A robot consists of two motors (differential drive), two microphones (left and right; we discuss their purpose later), a bump skirt, and one of the following additional configurations:

1. Three binary IR proximity sensors (left, right, and center)
2. Three binary IR proximity sensors, two light sensors (also binary)
3. Three binary IR proximity sensors, four robot detectors (these are able to determine if another robot is nearby in a certain direction; having four of these allows the robot to detect when a robot is ahead, behind, left, or right).
4. Four robot detectors (but no proximity sensors)

Each sensor (besides the microphones and bump skirt which are only processed by the instincts not the ENN directly) has its own input into the ENN and each motor its own output (hence configuration (2) has 5 inputs and 2 outputs). Each robot is equipped with the following instincts and reflexes:

1. Repulse Reflex: Moves the robot backwards and turns randomly
2. Crash Instinct: When the robot physically touches something, a negative reward is learned and the repulse reflex is triggered
3. Boredom Instinct: When the robot has been performing the same action for too long a negative reward is learned.
4. Sound to Left (respectively Right) Instinct: When the robot "hears" a clap to its left, it will move to the left (respectively right) randomly and add a positive reward for doing so.

We stress that, at the start of the simulation, the robot has really no sense of left or right, forward or backward. The "sound to left/right" instinct is provided so as to allow a human to train a robot to perform certain tasks (or just to help it along from time to time). It may seem strange that here we make a knowing distinction between left and right, however we justify it by observing that such instincts appear naturally in many organisms on this planet, so it is not such a stretch to use it here.

Using these configurations and instincts, we've discovered the following behaviors develop through multiple simulation runs:

- Configuration 1:

  i) Basic obstacle avoidance

  ii) Wall following

  iii) Searching for obstacles by scanning left/right every so often

- Configuration 2:

  i) Obstacle avoidance while seeking, avoiding, or ignoring light

- Configuration 3:

  i) Obstacle avoidance

  ii) "Follow-the-Leader" when at least one other robot was in the same simulation

  iii) Group together, or avoid each other

- Configuration 4:

  i) "Follow the leader" if another robot of configuration type 1-4 was in the simulation (hence following this other robot resulted in not crashing)

In (Ronald et al., 1999), the authors defined the emergence test as follows. Involving a system designer and an observer, the test proceeds in three stages:

1. Design: The system has been constructed by describing local elementary interactions between components in a language $\mathcal{L}_1$.
2. Observation: The observer (who knows $\mathcal{L}_1$) describes global behaviors and properties of the running system over time using a language $\mathcal{L}_2$.
3. Surprise: $\mathcal{L}_1$ is distinct from $\mathcal{L}_2$; furthermore the causal link between elementary interactions described in $\mathcal{L}_1$ and the behaviors actually observed in $\mathcal{L}_2$ is non-obvious to the observer (who is therefore surprised)

In our case, $\mathcal{L}_1$ is simply the three instincts along with the one reflex mentioned above. Interactions here are those that will maximize a robot's reward value. The only thing that may diminish the reward is crashing into a wall. The language $\mathcal{L}_2$ consists of those behaviors mentioned above.

Of course basic obstacle avoidance is an expected behavior (hence doesn't pass the emergence test). Wall following however was remarkable considering that the proximity

sensors used had no notion of distance (despite this some robots learned to follow the walls very closely; others from a greater distance). Also there is no reward for following a wall (only a reward for not touching it). Hence we claim this passes the test. We've also seen robots that learn to scan for walls by moving forward a certain amount then sweeping left then right. They used this strategy to follow walls closely. Again this behavior is emergent from the test.

We were also excited to see robots learning to follow each other or to group together in configuration 3. Again, there is no direct reward for doing so; hence this is an emergent behavior.

Configuration 4 learning to follow another robot (which is able to avoid walls) does not pass the emergent test (since following the other robot is the only expected strategy). Still it was an interesting result so we mention it here.

We note that the majority of the time, robots learn the very basic wall avoidance strategy. Also, robots seem to develop these emergent behaviors with or without human intervention (via the "clapping" instinct). Though it is interesting to note that more complicated behaviors seem (according to our simulations) more likely to develop if there is an occasional "helping hand" from a human supervisor (directing the robot away from a wall for example or when they get stuck in a corner; while the robot would eventually find its way out of such positions, the process is expedited by a few "claps").

Video recordings of some of these behaviors are available at our website: http://www.walterkrawec.org/robots/paper_alife13.html

Adapting to Change

We mention briefly that our robots seem to be very capable of adapting to change. While more work needs to be done investigating this area, we mention that a robot is able to compensate for a faulty sensor (e.g. a binary proximity sensor that is inverted). Also, when the sensor is restored, the robot is able to return to normal fairly quickly. All of this in real time while the robot is running. Of course compensating for faulty I/O is a quality shared by other neural networks, we were pleased to see that our STM architecture allowed for this compensation (instead of constantly enforcing old behaviors).

Additionally, we are able to add I/O "on the fly". This is accomplished simply by inserting extra I/O neurons (these are neurons that cannot be removed by the evolution algorithm) - after some iterations of the evolution process, assuming there is STM information for this new I/O, the neurons are incorporated into the network. We experimented with this by teaching a robot to use a collection of short-range proximity sensors then, when these have been learned, adding additional longer-range proximity sensors. The robot was able to learn to incorporate the new sensors while still maintaining the ability to use the original.

Applications of the Evolution Tape

We mention briefly some of the applications of the evolution tape. As already mentioned, it allows us to create a multi-threaded version of the ENN program thereby permitting one thread to work constantly on the evolution algorithm while another simply runs the produced ENN in real-time. We then use the evolution tape to synchronize the two threads' networks.

Secondly, it allows an ENN to run on embedded hardware with limited memory. The embedded device will, on occasion, request the latest evolve tape (or rather the updated section since its last request) from a PC. The PC is responsible for the actual evolution of the network. Furthermore, the embedded device will send to a PC whatever reward information it receives (via its instincts). Note that this is different from having a simple wireless network to the robot from which a PC both runs and evolves an ENN. By permitting an ENN to run locally on the robot itself, it may leave the range of the network, while also storing whatever reward information it receives (instincts should run locally on the embedded hardware). Then when back in range, the robot may send its reward information to the PC (which will incorporate it into its STM for fitness evaluations) and also request the latest evolve tape. This system does not require continual wireless communication.

We demonstrated this ability using a Parallax Propellor[1]. This is an 8 core (though our system currently uses only one core) MCU with 32KB of RAM and, on our prototype board, a 5MHz clock rate. Though much work needs to be done with this; it proves that an ENN may be run (relatively easily) on embedded hardware.

We are also very interested in designing a distributed evolution algorithm. In such a setup we will allow multiple computers to each have a copy of some network $\mathcal{N}$ and independently run a single iteration of the evolution algorithm (each computer shares a copy of the STM). When finished, the network with the highest fitness value will be chosen as the output. Whichever computer has this network will send its evolve tape to the others (again, only the portion listing the modifications made to the network which is at most $20(R+1)$ bytes, where R is the previously defined recursion level used) allowing each to easily be synchronized. Such a setup will permit us to explore a larger section of the solution space. Furthermore, we may then use the before mentioned technique to allow the network to be quickly transferred to embedded hardware and run in real time.

Summary and Future Work

In this paper, we experimented with controlling a robot using a real-time evolving neural network and observed that many of the behaviors that presented themselves passed the

[1] Propellor is a registered trademark of Parallax, of Rocklin, CA

emergence test of (Ronald et al., 1999). We argued that using a hill-climbing approach as opposed to the more typical genetic algorithm allows a robot to grow continually as an individual. That is, instead of replacing a robot's controller whenever a new population member is born, a robot slowly grows from itself. This is important for satisfying (Ross et al., 2003)'s checklist of conditions for allowing a robot to "grow up". Furthermore, a robot's past behavioral history (and personality "quirks") tends to be preserved through the evolution process.

We've already mentioned that our system satisfies conditions (1) and (4) of this checklist. Condition (3) is also satisfied since it is clear that a robot's learning is continual and we may teach a robot one skill at a time or even add new I/O to our robot who will then learn to use it. Condition (2) (that a robot learn a hierarchy of skills with new skills facilitated by older ones) we also believe is within our reach however more experimentation is required. Condition (5), which requires that the individual becomes capable of purposeful action on longer time spans, we think is also possible with our system however we have not yet constructed experiments that last long enough (we ended the majority of our simulations after 30 minutes of wall-clock time). This remains a future problem to investigate.

We also think that the EL-Learning system can be improved to take further advantage of the ENN's ability to easily memorize patterns in the received input. We believe the current mechanism doesn't promote this to its full potential at the moment.

Other work includes improving the efficiency of our learning algorithm and also to devise a distributed learning system. We also intend to take advantage of the ability to easily transfer an ENN onto embedded hardware to design a physical robot.

Finally we would like to experiment with the GA/HC hybrid approaches we mentioned before.

Acknowledgements

The author would like to thank the anonymous reviewers for their suggestions.

References

Cliff, D., Harvey, I., Husbands, P. (1992) Incremental evolution of neural network architectures for adaptive behaviour. *Technical report CSRP256. University of Sussex School of Cognitive and Computing Sciences.*

Floreano, D., Keller, L. (2010) Evolution of Adaptive Behaviour in Robots by Means of Darwinian Selection. *PLos Biol.* 8(1): e1000292. doi:10.1371/journal.pbio.1000292

Junfei, Q., Zhanjun, H., Xiaogang, R. (2007) Q-Learning based on neural network in learning action selection of mobile robot In *Automation and Logistics, 2007 IEEE International Conference*, pages 263–267. Jinan, China

Stanley, K. O., Miikkulainen, R. (2002) Efficient Reinforcement Learning through evolving network topologies In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, 9, San Francisco.

Stanley, K. O., Bryant, B. D., Miikkulainen, R. (2005) Real-time neuroevolution in the NERO video game *IEEE Transactions on Evolutionary Computation* 9(6), pages 652–668.

Ross, P., Hart, E., Lawson, A., Webb, A., Prem, E., Poelz, P., Morgavi, G. (2003). Requirements for getting a robot to grow up *Advances in Artificial Life Vol 7*. 7th European Conference, ECAL 2003, Dortmund, Germany. pages 847–856.

Ronald, E., Sipper, M., Capcarrére, M. (1999) Testing for emergence in artificial life In Floreano, D., Nicoud, J., Mondada, F., editors, *ECAL '99 Proceedings of the 5th European Conference on Advances in Artificial Life*, pages 13–20. Springer-Verlag, London.

Watkins C., Dayan, P. (1992) Q-Learning *Machine Learning*, Vol. 8. pages. 279–292.