Chapter 5
A Neuroevolutionary Approach to Adaptive Multi-agent Teams

Bobby D. Bryant and Risto Miikkulainen

5.1 Introduction

Multi-agent systems are a commonplace in social, political, and economic enterprises. Each of these domains consists of multiple autonomous parties cooperating or competing at some task. Multi-agent systems are often formalized for entertainment as well, with instances ranging from team sports to computer games. Games have previously been identified as a possible "killer application" for artificial intelligence [14], and a game involving multiple autonomous agents is a suitable platform for research into multi-agent systems as well.

The agents that comprise a multi-agent system can be either homogeneous or heterogeneous. Heterogeneous teams are often used for complex tasks because they allow agents to be specialized for sub-tasks (e.g. [2, 13, 29]). However, heterogeneous teams of sub-task specialists are brittle: if one specialist fails then the whole team may fail at its task. Moreover, when the agents in a team are programmed or trained to optimize a pre-specified division of labor, the team may perform inefficiently if the size of the team changes – for example, if more agents are added to speed up the task – or if the scope of the task changes.

For example, suppose you owned a team of ten reactor cleaning robots, and the optimal division of labor for the cleaning task required two sprayers, seven scrubbers, and one pumper (Fig. 5.1). If the individual robots were programmed or trained as sub-task specialists the team would be brittle and lacking in flexibility. Brittle, because the failure of a single spraying robot would reduce the entire team to half speed at the cleaning task, or the loss of the pumper robot would cause the team to fail entirely. Inflexible, because if a client requested a 20% speed-up for the task you would not be

B. D. Bryant (✉) · R. Miikkulainen
Department of Computer Sciences, University of Texas at Austin, Austin, USA
e-mail: bdbryant257@gmail.com

R. Miikkulainen
e-mail: risto@cs.utexas.edu

H. A. Abbass et al. (eds.), *Foundations of Trusted Autonomy*, Studies in Systems, Decision and Control 117, https://doi.org/10.1007/978-3-319-64816-3_5

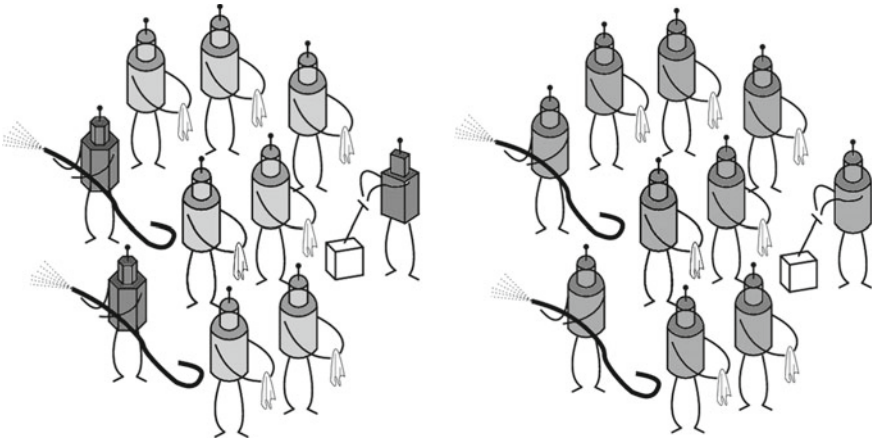

Fig. 5.1 *Left*: A heterogeneous team of cleaning robots is trained or programmed for sub-task specializations. Such a system is inflexible when the scope of the task changes, and brittle if key specialists are unavailable. *Right*: An Adaptive Team of Agents provides every agent with the capability of performing any of the necessary sub-tasks, and with a control policy that allows agents to switch between tasks at need. The resulting team is more flexible and less brittle than the heterogeneous team

able to simply send in 20% more robots; you would have to add ⌈20%⌉ more robots for each sub-task specialization, four more robots in all rather than two.

An alternative approach is to use a team of homogeneous agents, each capable of adopting any role required by the team's task, and capable of switching roles to optimize the team's performance in its current context. We call such a multi-agent architecture an *Adaptive Team of Agents* (ATA) [4]. An ATA is a homogeneous team that self-organizes a division of labor in situ so that it behaves as if it were a heterogeneous team. It changes the division dynamically as conditions change, and if composed of autonomous agents it must be able to organize the necessary divisions of labor without direction from a human operator.

Thus the ATA requires trusted autonomy. Within the team, individual agents must trust all the others to "do the right thing". Agents cannot select appropriate sub-tasks without some sort of assurance – possibly supported by observation – that the other members of the team are also selecting contextually appropriate sub-tasks. The owner of the team, whether in the context of robotics, simulation, or games, must also be able to trust the team as a whole to work out an effective division of labor in order to get the team's overall task done thoroughly and efficiently. That is, the team must pursue its owner's intent, and either the reality or appearance of intent may need to be instilled into the individual agents in order to achieve that. An ATA is robust because there are no critical task specialists that cannot be replaced by other members of the team; it is flexible because individual agents can switch roles whenever they observe that a sub-task is not receiving sufficient attention. If necessary, an agent can alternate between roles continuously in order to ensure that sufficient progress is made on all

sub-tasks. Thus for many kinds of task an ATA could be successful even if there are fewer agents than the number of roles demanded by the task.

Such adaptivity is often critical for autonomous agents embedded in games or simulators. For example, games in the *Civilization*™ genre usually provide a "settler" unit type that is capable of founding new cities plus carrying out various construction tasks. Play of the game requires a division of labor among the settlers, and the details of the division vary with the number in play and the demands of the growing civilization – e.g. the choice between founding more cities versus constructing roads to connect the existing cities. If the settler units were heterogeneous, i.e. each recruited to perform only one specific task, there would be a great loss of flexibility and a risk of complete loss of support for a game strategy if all the settlers of a given type were eliminated. But in fact the game offers homogeneous settlers, and human players switch them between tasks as needed. For embedded autonomous settlers, that switching would have to be made by the settler units themselves: an Adaptive Team of Agents is desirable.

Here we explore the Adaptive Team of Agents experimentally, using genetic algorithms to train artificial neural networks (ANN) as the "brains" for the agents in a game, and find that it is possible to evolve an ATA with ANN controllers for a simple but non-trivial strategy game. The game, called *Legion II*, is described in Sect. 5.2, and the agents' control architectures are described in Sect. 5.3. The evolutionary mechanism used to train the game agents to behave as an adaptive team, called *Neuroevolution with Enforced Sub-Populations* (ESP), is described in Sect. 5.4. Methodological considerations are addressed in Sect. 5.5, and then experiments are reported in Sect. 5.6. Finally, discussion of the experimental results and an examination of future directions is given in Sect. 5.7.

5.2 The *Legion II* Game

Legion II is a discrete-state strategy game designed as a test bed for studying the division of labor in multi-agent systems. It requires a group of legions to defend a province of the Roman Empire against the pillage of a steady influx of barbarians. The legions are the agents under study; they are trained by the method described in Sect. 5.4, or by other methods reported elsewhere (e.g. [3, 5, 6]). The barbarians act according to a preprogrammed policy, to serve as a foil for the legions.

Legion II provides challenges similar to those provided by other games and simulators currently used for computational intelligence approaches to multi-agent learning research (e.g. [1, 15, 26, 27]), and is expandable to provide more complex learning challenges as research progresses. In its current incarnation it is incrementally more complex than a multi-predator/multi-prey game. It is conceptually similar to the pseudo-*Warcraft*™ simulator used in [1], differing primarily in its focus on the predators rather than the prey, and consequently in the details of scoring games.

The following subsections describe the components, features, and rules of the *Legion II* game, including the map, the game agents, and the method of calculating game scores.

5.2.1 The Map

Legion II is played on a planar map. The map is tiled with hexagonal cells in order to discretize the location and movement of objects in the game; in gaming jargon such cells are called *hexes* (singular *hex*). Several randomly selected map cells are distinguished as *cities*, and the remainder are considered to be *farmland* (Fig. 5.2).

The hex tiling imposes a six-fold radial symmetry on the map grid, defining the cardinal directions NE, E, SE, SW, W, and NW. This six-fold symmetry, along with the discretization of location imposed by the tiling, means that an agent's atomic moves are restricted to a discrete choice of jumps to one of the six cells adjacent

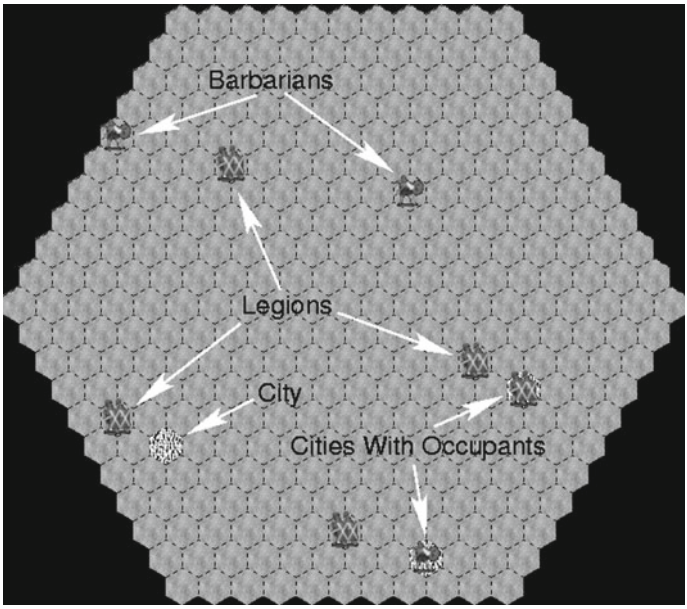

Fig. 5.2 A large hexagonal playing area is tiled with smaller hexagons in order to discretize the positions of the game objects. Legions are shown iconically as close pairs of men ranked behind large rectangular shields, and barbarians as individuals bearing an axe and a smaller round shield. Each icon represents a large body of men, i.e. a legion or a warband. Cities are shown in white, with any occupant superimposed. All non-city map cells are farmland, shown with a mottled pattern. The game is a test bed for multi-agent learning methods, whereby the legions must learn to contest possession of the playing area with the barbarians. (Animations of the *Legion II* game can be viewed at http://nn.cs.utexas.edu/keyword?ATA.)

to the agent's current location, and that the atomic move is always in one of the six cardinal directions. This map structure has important consequences for the design of the sensors and controllers for the agents in the game, which are described in detail in Sect. 5.3.

5.2.2 Units

There are two types of autonomous agents that can be placed on the map in a *Legion II* game: *legions* and *barbarians*. In accordance with gaming jargon these mobile agents are called *units* (singular *unit*) when no distinction needs to be made between the types.

Each unit is considered to be "in" some specific map cell at any time. A unit may move according to the rules described below, but its moves occur as an atomic jump from the cell it currently occupies to an adjacent one, not as continuous movement in Euclidean space.

The current position of each unit is shown by a sprite on the game map. In accordance with the jargon of the *Civilization* game genre, the sizes of the units are ambiguous. Thus the unit type called "a legion" represents a body of legionnaires, but is shown graphically as only a pair of men behind large rectangular shields. Similarly, the unit type called "a barbarian" represents a body of barbarians operating as a warband, but is shown graphically as only a single individual with axe and shield (Fig. 5.2).

The legions start the game already on the map, in randomly selected map cells. There are no barbarians in play at the start of the game. Instead, barbarians enter at the rate of one per turn, in a randomly selected unoccupied map cell.

5.2.3 Game Play

Legion II is played in turns. At the beginning of each turn a new barbarian is placed at a random location on the map. If the randomly generated location is already occupied by a unit, a new location is generated. This search continues until an unoccupied location for the new barbarian is found.

Thereafter, each legion is allowed to make a move, and then each barbarian in play makes a move. A unit's move can be a jump to one of the six adjacent map cells, or it can elect to remain stationary for the current turn. When all the units have made their moves the turn is complete, and a new turn begins with the placement of another new barbarian. Play continues for a pre-specified number of turns, 200 in all the experiments reported here.

All the units are autonomous; there is no virtual player that manipulates them as passive objects. Whenever it is a unit's turn to move, the game engine calculates the activation values for that unit's egocentric sensors, presents them to the unit's

controller, and implements the choice of move signaled at the output of the unit's controller, as described in Sect. 5.3.

There are some restrictions on whether the move requested by a unit is actually allowed, and the restrictions vary slightly between the legions and the barbarians. The general restrictions are that only one unit may occupy any given map cell at a time, and no unit may ever move off the edge of the playing area defined by the tiling.

If a legion requests a move into an unoccupied map cell, or requests to remain stationary for the current turn, the request is immediately implemented by the game engine. If the legion requests moving into a map cell occupied by another legion, or requests a move off the edge of the map, the game engine leaves the legion stationary for that turn instead. If the legion requests moving into a map cell occupied by a barbarian, the game engine immediately removes the barbarian from play and then moves the legion as requested.

If a barbarian requests a move that is neither off-map nor into an occupied map cell, the request is immediately implemented by the game engine. If the barbarian requests a move into a map cell occupied by either a legion or another barbarian, the game engine leaves it stationary for the current turn. (Notice that this does not allow a barbarian to eliminate a legion from play the way a legion can eliminate a barbarian.) If the barbarian requests a move off the edge of the map, the game engine consults the barbarian's controller to see what its second choice would have been. If that second choice is also off-map then the game engine leaves the barbarian stationary for the current turn; otherwise, the secondary preference is implemented.

Barbarians are given their second choice when they request a move off the map, because their programming is very simple, and it is not desirable to leave them 'stuck' at the edge of the map during a game. Legions do not get the second-chance benefit; they are expected to learn to request useful moves.

5.2.4 Scoring the Game

The game score is computed as follows. The barbarians accumulate points for any pillaging they are able to do, and the legions excel by minimizing the amount of pillage points that the barbarians accumulate. At the end of every game turn, each barbarian in play receives 100 points for pillage if it is in a city, or only a single point otherwise. The points are totaled for all the barbarians each turn, and accumulated over the course of the game. When a barbarian is eliminated no points are forfeited, but that barbarian cannot contribute any further points to the total thereafter.

This scoring scheme was designed in order to force the legions to learn two distinct classes of behavior in order to minimize the barbarian's score. Due to the expensive point cost for the cities, the legions must keep the barbarians out of them, which they can easily do by garrisoning them. However, further optimization requires any legions beyond those needed for garrison duty to actively pursue and destroy the barbarians in the countryside. If they fail to do so, a large number of barbarians will accumulate in the countryside, and though each only scores one point of pillage per

turn, their cumulative aggregate is very damaging to the legions' goal of minimizing the barbarian's score.

In principle the legions might be able to minimize the pillage by neglecting to garrison the cities and utilizing every legion to try to chase down the barbarians, but the random placement of the incoming barbarians means that they can appear behind the legions, near any ungarrisoned cities, and inflict several turns of the very expensive city pillaging before a legion arrives to clear them out. The barbarian arrival rate was by design set high enough to ensure that the legions cannot mop them up fast enough to risk leaving any cities ungarrisoned. Thus the legions *must* garrison the cities in order to score well, and any improvement beyond what can be obtained by garrisoning the cities can only come at the cost of learning a second mode of behavior, pursuing the barbarians.

For the purposes of reporting game scores the pillage points collected by the barbarians are normalized to a scale of [0, 100], by calculating the maximum possible points and scaling the actual points down according to the formula:

$$Score_{\text{reported}} = 100 \times Points_{\text{actual}}/Points_{\text{possible}} \tag{5.1}$$

The result can be interpreted as a pillage rate, stated as a percentage of the expected amount of pillaging that would have been done in the absence of any legions to contest the barbarians' activities. Notice that from the legions' point of view, *lower* scores are better.

In practice the legions are never able to drive the score to zero. This fact is due in part to the vagaries of the starting conditions: if the random set-up places all the legions very distant from a city and a barbarian is placed very near that city on the first turn, there is nothing the legions can do to beat that barbarian into the city, no matter how well trained they are. However, a factor that weighs in more heavily than that is the rapid rate of appearance of the barbarians versus the finite speed of the legions. Since the legions and barbarians move at the same speed, it is difficult for the legions to chase down the barbarians that appear at arbitrary distances away. Moreover, as the legions thin out the barbarians on the map the average distance between the remaining barbarians increases, and it takes the legions longer to chase any additional barbarians down. Thus even for well-trained legions the game settles down into a dynamic equilibrium between the rate of new barbarian arrivals and the speed of the legions, yielding a steady-state density of barbarians on the map, and thus a steady-state accumulation of pillage counts after the equilibrium is achieved.

5.3 Agent Control Architectures

The legions and barbarians are controlled by policies that map egocentric sensory inputs onto a choice of the discrete actions allowed in the game. This section describes their sensors and controllers. The simpler sensors and controllers used by the barbarians are described first, then the more elaborate system used to control the legions.

5.3.1 Barbarian Sensors and Controllers

The legions are the only learning agents in the game, so the barbarians can use any simple pre-programmed logic that poses a suitable threat to the legions' interests. The barbarians' basic design calls for them to be attracted toward cities and repulsed from legions, with the attraction slightly stronger than the repulsion, so that the barbarians will take some risks when an opportunity for pillaging a city presents itself. This behavior is implemented by algebraically combining two "motivation" vectors, one for the attraction toward cities and one for the repulsion from legions:

$$\mathscr{M}_{\text{final}} = \mathscr{M}_{\text{cities}} + 0.9\mathscr{M}_{\text{legions}} \tag{5.2}$$

Each vector consists of six floating point numbers, indicating the strength of the barbarian's "desire" to move in each of the six cardinal directions. The 0.9 factor is what makes the motivation to flee the legions slightly weaker than the motivation to approach the cities. After the combination, the peak value in the $\mathscr{M}_{\text{final}}$ vector indicates which direction the barbarian "most wants" to move. In situations where a second choice must be considered, the second-highest value in $\mathscr{M}_{\text{final}}$ is used to select the direction instead.

The values in the two arrays are derived, directly or indirectly, from the activation values in a simple sensor system. The barbarian's sensor system consists of two sensor arrays, one that detects cities and another that detects legions. Each array divides the world into six 60° non-overlapping egocentric fields of view. The value sensed for each field is:

$$s = \sum_i \frac{1}{d_i}, \tag{5.3}$$

where d_i is the distance to an object i of the correct type within that field of view. The distances are measured in the hex-tile equivalent of Manhattan distance, i.e. the length of the shortest path of map cells from the viewer to the object, not counting the cell that the viewer itself is in (Fig. 5.3).

For simplicity, if an object is exactly on the boundary between two fields of view, the sensors report it as being in the field to the clockwise of the boundary. Due to the relatively small map, no limit is placed on the range of the sensors.

Notice that this sensor architecture obscures a great deal of detail about the environment. It does not give specific object counts, distances, or directions, but rather only a general indication of how much opportunity or threat the relevant class of objects presents in each of the six fields of view.

Once these values have been calculated and loaded into the sensor arrays, the activations in the array that senses cities can be used directly for the $\mathscr{M}_{\text{cities}}$ vector in Eq. 5.2. $\mathscr{M}_{\text{legions}}$ can be derived from the values in the array that senses legions by permuting the values in the array to reverse their directional senses, i.e. the sensor activation for legions to the west can be used as the motivation value for a move to the east, and similarly for the other five cardinal directions. After the conversions

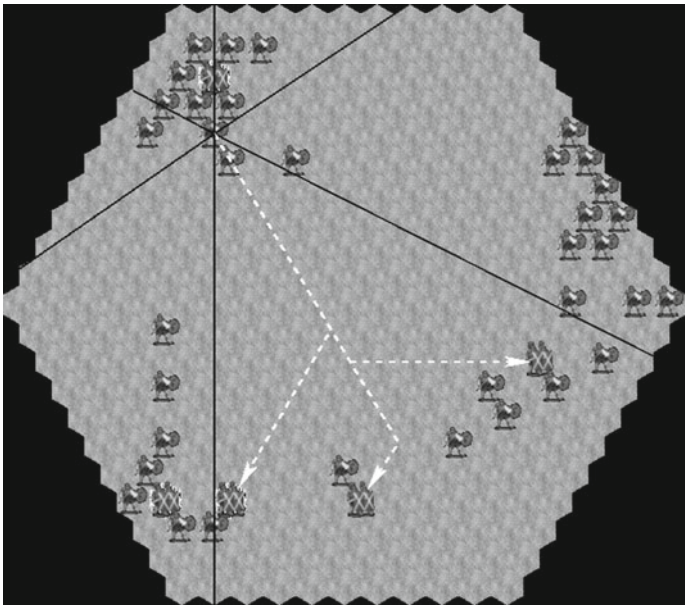

Fig. 5.3 The solid black lines show the boundaries of the six sensory fields of view for one barbarian near the northwest corner of the map. The boundaries emanate from the center of the map cell occupied by the barbarian and out through its six corners. The dashed white lines show the hexagonal Manhattan distances to the three legions in the SE field of view of the sensing barbarian. These lines are traced from the center to center along a path of map cells, and thus emanate through the sides of the hexagons rather than through the corners as the field boundaries do

from sensor activations to motivation vectors, $\mathscr{M}_{\text{final}}$ can be calculated and its peak value identified to determine the requested direction for the current move.

There is no explicit mechanism to allow a barbarian to request remaining stationary for the current turn. For simplicity the game engine examines a barbarian's location at the start of its move and leaves the barbarian stationary if it is already in a city. Otherwise the game engine calculates the values to be loaded into the barbarian's sensors, performs the numerical manipulations described above, and implements the resulting move request if it is not prohibited by the rules described in Sect. 5.2.3.

The resulting behavior, although simple, has the desired effect in the game. As suggested by Fig. 5.3, barbarians will stream toward the cities to occupy them, or congregate around them if the city is already occupied. Other barbarians will flee any roving legions, sometimes congregating in clusters on the periphery of the map. The barbarians are quick to exploit any city that the legions leave unguarded. They do, however, tend to get in each other's way when a legion approaches a crowd and they need to flee, resulting in many casualties, but that is perhaps an appropriate simulation of the behavior of undisciplined barbarians on a pillaging raid.

5.3.2 Legion Sensors and Controllers

Unlike the barbarians, the legions are required to learn appropriate behavior for their gameplay. They are therefore provided with a more sophisticated, trainable control system. The design includes a sensor system that provides more detail about the game state than the barbarians' sensors do, plus an artificial neural network "brain" to map the sensor inputs onto a choice of actions.

5.3.2.1 The Legions' Sensors

The legions are equipped with sensor systems that are conceptually similar to the barbarians', but enhanced in several ways. Unlike the barbarians, the legions have a sensor array for all three types of object in the game: cities, barbarians, and other legions. Also unlike the barbarians, each of those three sensor arrays are compound structures consisting of two six-element sub-arrays plus one additional element (Fig. 5.4), rather than the barbarian's simple six-element sensor arrays.

An array's two six-element sub-arrays are similar to the barbarians' sensor arrays, except that one only detects objects in adjacent map cells and the other only detects objects at greater distances. For the former, the game engine sets the array elements to 1.0 if there is an object of the appropriate type in the adjacent map cell in the appropriate direction, and to 0.0 otherwise. For the latter, the game engine assigns values to the elements slightly differently from the way it assigns values to the barbarian's sensors. First, it ignores objects at $d = 1$, since those are detected by the short-range array described above. Second, since the distances used in the calculations for this array are always greater than one, it deducts one from the distances used in the calculations, in order to increase the signal strength. That is, Eq. 5.3 used for the barbarians becomes:

$$s = \sum_i \frac{1}{d_i - 1}, \tag{5.4}$$

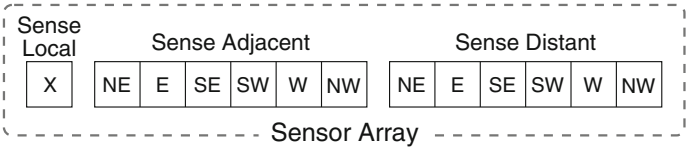

Fig. 5.4 The legions have three sensor arrays, one each for cities, barbarians, and other legions. Each of those three arrays consists of three sub-arrays as shown above. A single-element sub-array (left) detects objects co-located in the map cell that the legion occupies. Two six-element sub-arrays detect objects in the six radial fields of view; one only detects adjacent objects, and the other only detects objects farther away. These 13 elements of each of the three compound arrays are concatenated to serve as a 39-element input activation for an artificial neural network that controls the legion's behavior (Fig. 5.5)

for objects *i* of the correct type *and* at a distance greater than one, within that field of view.

A third difference is that sensed objects near a boundary between two sensor fields are not arbitrarily assigned to the field to the clockwise of the boundary. Instead, objects within 10° of the boundary (from the legion's perspective) have their signal split between the two fields. As a result each sensor array is effectively a set of 40° arcs of unsplit signals alternating with 20° arcs of split signals, though the aggregations result in an array of only six activation values. As with the barbarians' sensors, there is no range limit on this long-range sub-array.

The additional single sensor element in a compound array detects objects in the legion's own map cell: if an object of the appropriate type is present the game engine sets the value of this sensor to 1.0; otherwise it sets it to 0.0. However, a legion does not detect itself, and since the rules prevent multiple units from occupying the same map cell, the only time this local detection sensor is activated in practice is when the legion occupies a city. In principle the detect-local sensor could have been eliminated from the sensor arrays used to detect legions and barbarians, but identical arrays were used for all object types in order to simplify the implementation, and to make allowance for future game modifications that would allow "stacking" multiple units within a single map cell.

The full architecture of the compound sensors is shown in Fig. 5.4. The two sub-arrays contain six elements each, corresponding to the six cardinal directions. Thus together with the additional independent element, each array reports 13 floating point values ≥ 0.0 whenever a sense is collected from the environment. Since there is one compound sensor for each of the three types of game object, a legion's egocentric perception of the game state is represented by 39 floating point numbers.

5.3.2.2 The Legions' Controller Network

A legion's behavior is controlled by a feed-forward neural network. The network maps the legion's egocentric perception of the game state onto a choice of moves. Whenever it is a legion's turn to move, the game engine calculates the sensor values for the legion's view of the current game state and presents the resulting 39 floating point numbers to the input of the controller network. The values are propagated through the network, and the activation pattern at the network's output is decoded to determine the legion's choice of move for the current turn (Fig. 5.5).

The output layer of the networks consist of seven neurons, corresponding to the seven discrete actions available to the legions. When the input activations have been propagated through the network the activation pattern at the output layer is interpreted as an *action unit coding*, i.e. the action corresponding to the output neuron with the highest activation level is taken to be the network's choice of action for the current turn.

In addition to the sensory inputs, each neuron in the controller networks is fed by a bias unit with a fixed activation of $+1.0$ and a trainable weight to propagate the value into the neuron's accumulator. For the experiments reported below, the

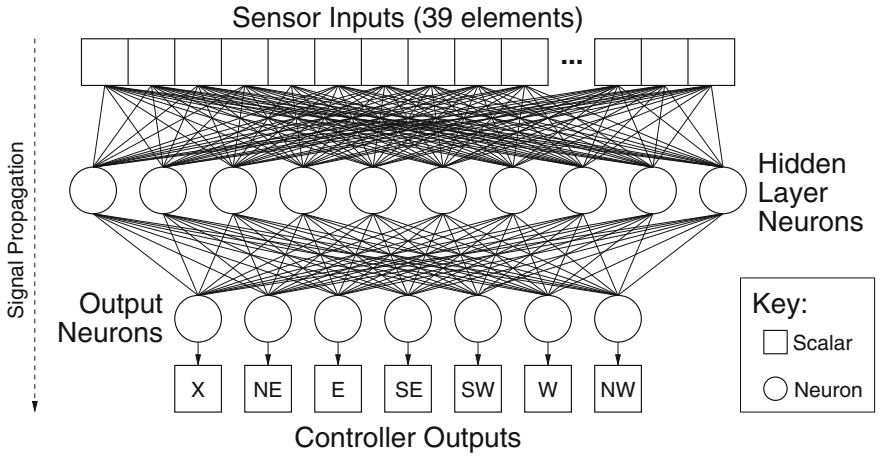

Fig. 5.5 During play the values obtained by a legion's sensors are propagated through an artificial neural network to create an activation pattern at the network's output. This pattern is then interpreted as a choice of one of the discrete actions available to the legion. When properly trained, the network serves as a "brain" for the legion as an autonomous agent in the game

controller network's hidden layer consisted of 10 neurons, which was found to be effective in preliminary survey experiments.

5.3.2.3 Properties of the Legions' Control Architecture

There are a number of important consequences of the adoption of this sensor/controller architecture for the legions, which the reader may wish to keep in mind while reading about the methodologies and experiments:

- *The sensor readings are egocentric*. For a given state of the game, each of the legions in play will perceive the map differently, depending on their individual locations on the map.
- *The sensors provide a lossy view of the map*. The legions have complete state information about their immediate neighborhood, but that is reduced to a fuzzy "feel" for the presence of more distant objects.
- *The legions must work with uninterpreted inputs*. There is a semantic structure to the sensor arrays, but that structure is not known to the legions: the sense values appear as a flat vector of floating point numbers in their controller networks' input layers. The significance of any individual input or set of inputs, or of any correlations between inputs, is something the legions must obtain via the learning process.
- *There is no explicit representation of goals*. None of the network inputs, nor any other part of the controller logic, provide a legion with any sort of objective.

Coherent higher-level behavior must be learned as a response to a sequence of inputs that vary over time.

- *The legions do not have memory*. The feed-forward controller networks do not allow for any saved state, so the legions are not able to learn an internal representation for goals to make up for the lack of externally specified goals. All actions are immediate reactive responses to the environment.

These various requirements and restrictions conspire to present the legions with a very difficult challenge if they are to learn to behave intelligently. The intent of the game designer, and any real or apparent intent on the part of the individual legions, must be instilled by means of the learning system. However, experience shows that properly trained artificial neural networks excel at producing the appearance of purposeful intelligent behavior (e.g. [1, 10, 17, 19, 20, 23]).

5.4 Neuroevolution With Enforced Sub-Populations (ESP)

For many agent control tasks the correct input-output mappings for the agents' controllers are not known, so it is not possible to program them or train them with supervised learning methods. However, controllers such as artificial neural networks can be evolved to perform a task in its actual context, discovering optimal mappings in the process. The use of a genetic algorithm to train an artificial neural network is called *neuroevolution*. Surveys of the field can be found in [24, 28]. An overview of the use of neuroevolution to learn egocentric input-output mappings for game agents' controllers can be found in [16].

One of the most empirically effective neuroevolutionary algorithms yet devised is *Neuroevolution with Enforced Sub-Populations* (NE-ESP, or usually just ESP) [9, 11]. The basic concept behind ESP is that each genetic representation specifies only a single neuron rather than an entire network, and a separate breeding population is maintained for each neuron in the network.

Evaluations cannot be made on a network's neurons in isolation, so the evaluations in ESP are done by drawing one neuron at random from each sub-population, assembling them into a complete network, evaluating the network as for any other neuroevolutionary algorithm, and ascribing that network's fitness score back to each of the individual neurons used to create it. When all the neurons in all the populations have been evaluated, selection and breeding is done independently within each sub-population. However, the fitness of an individual neuron depends not on its properties in isolation, but on how well it works together with neurons from the other populations. Thus the neurons in the sub-populations are subjected to cooperative coevolution [18, 21], and as evolution progresses they converge as symbiotic species into functional niches that work together in a network as a good solution to the target problem.

ESP was originally introduced for training fully recurrent networks as continuous-state controllers, e.g. for the inverted pendulum problem and the conceptually similar

application to a finless rocket [12]. Both the application and the details of the ESP implementation used for *Legion II* are novel.

In *Legion II* ESP is used for learning to make a discrete choice among the legions' possible atomic actions. For the experiments reported here, the controller networks were non-recurrent feed-forward networks with a single hidden layer, as described in Sect. 5.3.2.2 (Fig. 5.5). A distinct sub-population was used for each position for a neuron in the network, regardless of which layer it is in; the representations in the populations held only the weights on the input side of the neurons (Fig. 5.6). In principle it is not necessary to provide separate neurons for the output layer; an architecture more similar to previous uses of ESP would have dispensed with those neurons and stored both the input and output weights in the representations of the hidden-layer neurons. That is in fact the mechanism used in the original *Legion I* experiments [4]. However, the introduction of new sub-populations for the output layer contributed to the improved scores in the experiments reported here.

Fitness evaluations were obtained during evolution by playing the current generation of controllers against randomly generated game setups; the set of possible game setups is so large that none ever have to be reused. A different sequence of training games was used for each independent run with a given parameterization in a given experiment. For fair evaluations within a single run, every neuron was evaluated against the same game before moving on to the next game. The methodology is described in more detail in Sect. 5.5.

When the ESP mechanism is used, the actual fitness of a network is ascribed to each neuron used to construct it. As a result, the ascribed fitness is only an estimate of a neuron's "true" fitness; the "true" fitness is in fact ill-defined, since the neurons are only useful when associated with other neurons in the other populations. However, a reasonable estimate of the fitness of a neuron – *given that it will be used in a network with neurons from the other populations* – can be obtained by evaluating the neuron repeatedly, in networks comprised of independent random selections of neurons.

Thus for the experiments described here each neuron was evaluated on three different games per generation, and the three resulting fitness ratings were averaged to estimate the neuron's fitness. The associations of the neurons into networks were re-randomized before each of the three games so that the averaged fitness ratings would reflect the quality of a given neuron per se more than the quality of the other neurons it happened to be associated with in the network. Each of the three evaluations used a different game setup, and all of the neurons were evaluated on the same three games during the generation.

Since the training game setups differed continually from generation to generation, learning progressed somewhat noisily: a neuron that performed well on the training games in one generation might not perform well on the new training games of the following generation. However, neuroevolution with ESP is robust even when evaluations are somewhat noisy, and the use of three games per generation helped smooth the noise of the evaluations. The continually changing stream of training games from generation to generation required candidate solutions to generalize to novel game setups, or else risk having their constituent neurons be weeded out of the breeding population; if a network performed poorly on the game setup used during a given

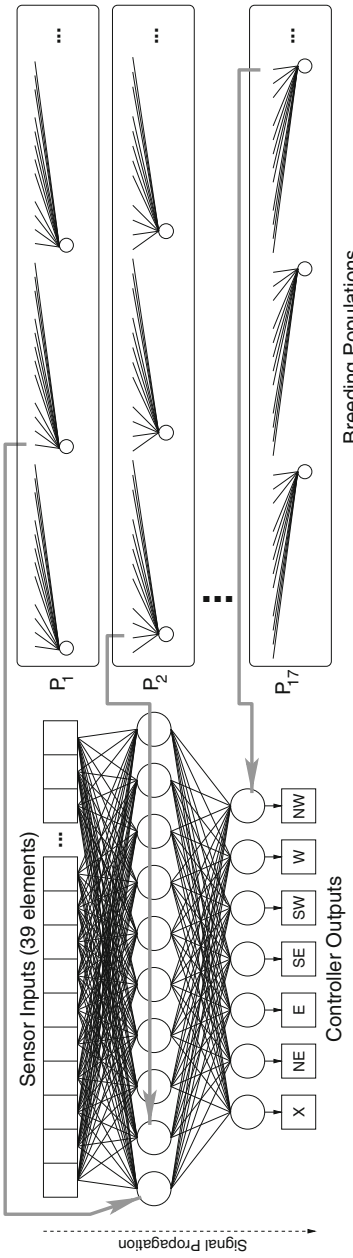

Fig. 5.6 To apply the ESP method of neuroevolution for training the legions' controllers, a separate breeding population was maintained for each of the 17 neurons used in the controller network

generation it received a poor fitness score, regardless of how well it had performed during previous generations.

As described in Sect. 5.5.2, an evaluation against the validation set was done at the end of every generation. For ordinary evolutionary mechanisms the network that performed best on the current generation's fitness evaluations would be chosen for the run against the validation set. However, the notion of "best network in the population" is ill-defined when the ESP mechanism is used, so for the *Legion II* experiments a *nominal best network* is defined as the network composed by selecting the most fit neuron from each sub-population. It was that nominal best network that was evaluated against the validation set at the end of each generation.

Breeding was done by a probabilistic method that strongly favored the most fit solutions, but also allowed less fit solutions to contribute to the next generation with low probability. The mechanism was as follows. When all the training evaluations for a generation were complete, the storage for the representations of the solutions in each sub-population was sorted from most fit to least fit, so that the most fit had the lowest index. Then each representation was replaced one at a time, starting from the highest index (i.e., the least fit neuron in the population). Two parents were selected with uniform probability over the indices less than or equal to the index of the representation currently being replaced. I.e., that representation or any more fit representation could be chosen as a parent. The two selections were made independently, so that it was possible for the same representation to be used for both parents; in such cases the child would differ from the parent only by mutations. (Notice that this mechanism *always* breeds the most fit neuron with itself at the final pairing.) Since the less fit representations were progressively eliminated from the effective breeding pool, the more fit solutions had more opportunities to contribute to the next population. Preliminary survey experiments showed that this mechanism produced better results than a simple elitist mechanism.

Once a pair of parents were selected they were bred with either 1-point or 2-point crossover, with a 50% chance for each. Only one child was produced from the crossover; the remaining genetic material was discarded. Each weight in the representation was then subjected to a mutation at a 10% probability, independently determined. Mutations were implemented as a delta to the current weight chosen from the exponential distribution (Eq. 5.5) with $\lambda = 5.0$, and inverted to be a negative delta with a 50% chance.

$$f(x, \lambda) = \lambda e^{-\lambda x}, x \geq 0 \tag{5.5}$$

That choice of λ reduced the mean of the distribution, and was chosen on the basis of preliminary survey experiments. The deltas resulting from this distribution were small with high probability, but potentially very large with a low probability. That distribution allowed mutations to support both fine tuning of the weights and jumps to more distant regions of the solution space.

Training on the *Legion II* problem with neuroevolution makes progress asymptotically. For the experiments reported here, evolution was allowed to continue for 5000 generations, well out onto the flat of the learning curve, to ensure that comparisons and analyses were not made on undertrained solutions.

5.5 Experimental Methodology

The *Legion II* experiments followed the familiar methodology of using distinct training, validation, and test sets. However, procedural questions arise when applying that methodology to a game such as *Legion II*. This section explains how those questions were resolved for the experiments reported below.

5.5.1 Repeatable Gameplay

When training or testing by means of dynamic gameplay rather than static examples, it is useful to have a definition for the concept of "the same game", e.g. to make comparative evaluations of the performance of embedded game agents. However, games that are genuinely identical with respect to the course of play are, in general, impossible to generate, if they involve embedded game agents that learn: as the agents learn, their behavior will change, and the changed behavior will cause the course of the game to vary from earlier plays. For example, if the legions in *Legion II* fail to garrison the cities during the early stages of training, the barbarians will occupy the cities. But later during training, when the legions have learned to garrison the cities, the details of the barbarians' behavior must also change in response – i.e., the city will not be pillaged as before – even if there has been no change to the starting state and the barbarians' control policy.

It is therefore useful to have a pragmatic definition of "the same game" for experimental work. Thus for *Legion II* two games are identified as "the same game" if they use the same starting position for the cities and legions, and the same schedule for barbarian arrivals. The schedule for arrivals includes both the time and the randomly selected position on the map. For all the games reported here the barbarian arrivals were fixed at one per turn, so only their placement mattered for identifying two games as being the same.

However, the randomized placement of the barbarians is not always repeatable: as described in Sect. 5.2.3, if the position selected for placing a new barbarian on the map is occupied, an alternative randomly selected position is used instead, and re-tries continue until an empty map cell is found. But as described above, changes to the legions' behavior will result in different game states at a given point in time for various instances of "the same game", so a barbarian placement during one play of the game may not be repeatable in another run using a different controller for the legions. Therefore, for pragmatic reasons, "the same game" is defined for *Legion II* to consider *only the first try* for the positioning of arriving barbarians; the additional tries triggered by the unavoidable divergences of the game state are not considered to make two games different.

This concept of "the same game" was used to create sets of games that were used repeatedly during training and testing, as follows.

5.5.2 Training

Randomized learning algorithms such as neuroevolution do not always produce their best solution at the end of a fixed-length run; the random modifications to the representations in an evolutionary population can make the solutions worse as well as better. Therefore it is useful to have a mechanism for returning the best solution obtained at any time in the course of a run.

The commonly used mechanism is to evaluate candidate solutions periodically during training, and, if the top performer is better than any previously encountered during the run, to save that top performer as the potential output of the learning algorithm. At the end of the run, the most recently saved top performer is returned as the solution produced by the algorithm. The learning algorithm still *runs* for some fixed number of iterations that the experimenter deems sufficient for finding a good solution, but that solution may be discovered at any time during the run.

The periodic evaluation is performed against a *validation set*. When generalization is desired, the validation set must be independent of the training data; otherwise the algorithm will return a solution that is biased toward good performance on the training data at the expense of poorer performance on more general data of the same type. For supervised learning, the validation set normally takes the form of a reserved subset of the available training examples. However, when annotated examples are not available, such as when using evolutionary learning to learn a motor control task or a controller for an embedded game agent, the validation set can be a standardized set of example problems. The definition of "the same game" in *Legion II* allows construction of a distinctive set of games to serve as the validation set for *Legion II* learning tasks, and that is the mechanism used in the experiments reported here.

Therefore stopping was handled in the experiments by running the learning algorithms for a period deemed to be "long enough", and using the validation set mechanism to control which candidate was actually returned as the result of a run. The validation set for *Legion II* was a set of ten games. A set of ten games with independently generated starting positions and barbarian placement positions was judged to be a sufficient evaluation for generalization; larger sets adversely affect the run time of the evolutionary algorithm. The score for a controller's validation performance was defined as the average of the game scores obtained by play against the ten games of the validation set.

An evaluation was made against the validation set at the end of each generation, and the nominal best network saved if its validation score was better than at any previous generation. For a given run of the training program the same validation set was used at each evaluation period, to ensure consistent evaluations. However, the validation set was created independently for each run. The idea is that each run should represent an independent sample of the space of all possible runs, for a given parameterization of the learning algorithm. Since the random selection of a validation set is part of the the "possible world" of the run of a stochastic algorithm, its construction was allowed to vary from run to run, along with all the other stochastic decisions.

5.5.3 Testing

Each run of the learning algorithm returned a single neural network as its output. The networks were saved to files for later testing with a separate program; the test program spilled various run-time metrics to a file for analysis and plotting with the *R* statistical computing environment [22].

Tests of current performance were also conducted during the course of training, for the production of learning curves. Tests were only run on those generations where the evaluation on the validation set produced a new top performer, i.e. when measurable progress had been made. These were "side tests"; the learning algorithms ran the tests and spilled the results to a file for later analysis, but did not make any training decisions on the basis of the tests, to avoid biasing the training toward the test games.

Whether during the learning run or afterward, tests were run on a set of games constructed as the validation set was, but independent of both the validation set and the training games. As with the validation evaluations, the evaluation score for this composite test was defined as the average of the scores obtained on the individual games of the test set.

Unlike the validation set, the same test set was used for every independent run of every learning algorithm, to ensure that any differences in the test metrics were the result of differences in the solutions being examined rather than differences in the difficulty of independently generated test sets. The training-time evaluations on the test set are not as frequent as the evaluations on the validation set, so a larger test set could be used without unduly extending the run times of the training algorithms. Also, it is essential that the test set be an accurate model of the set of possible games; therefore a set of 31 games was used. (Statisticians deem a minimum of 30 samples necessary for characterizing a distribution when the measurements are not known *a priori* to fall into a normal distribution; it sometimes proves useful to use 31 rather than that minimum, so that there will be a clearly defined median for any measurement, to be used as a principled choice whenever it proves useful to plot or analyze a single "typical" example.)

5.6 Experiments

ANN controllers for the legions in *Legion II* were trained using ESP and the procedures described above. The homogeneity required by the ATA architecture was enforced by using the same controller to make all the legions' decisions during a game. The game parameters were set to require a division of labor to perform well: there were more legions than cities, the randomized placement of the barbarians and the 100:1 ratio of pillage between the cities and countryside made it essential to garrison the cities, and the large number of barbarians arriving over the course of the game made it essential to eliminate barbarians in the countryside as well, if pillage was to be minimized. With one barbarian arriving per turn, the count would ramp up

from one to 200 over the course of a game in the absence of action by the legions, providing an average of ~100 pillage points per turn. With three cities each subject to an additional 100 pillage points per turn, pillaging the countryside can amount to ~1/4 of the worst possible score. Thus the legionary ATA must take actions beyond simply garrisoning the cities in order to minimize the pillage: a division of labor is required.

The following sections examine the results of the training experiment and the behavior produced in the legions.

5.6.1 Learning the Division of Labor

Hundreds of runs of neuroevolutionary learning on the *Legion II* problem, with a variety of learning parameters and a number of changes to the game rules and network architecture since the initial results reported in [4], have consistently performed well, where "well" is defined fuzzily as "learns to bring the pillage rate substantially below the 25% threshold" obtainable by a policy of static garrisons and no division of labor to support additional activity by the spare legions. For the experiments reported here, eleven independent runs of the base learning method with the parameters described in Sect. 5.4 (but independent streams of random numbers) produced a mean test performance score of 4.316%, with all falling in the range 3.5–6.0%. (Recall that there is no *a priori* expectation that a 0% pillage rate could be learned.) The scores on the games in the test set show that all eleven runs produced controllers that allowed the legions to reduce pillaging well below the 25% rate obtainable by garrisoning the cities and taking no further actions against the barbarians.

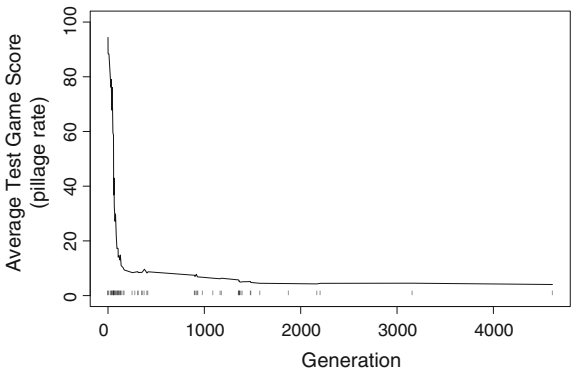

Fig. 5.7 The plot shows progress against the test set for the median performer of the eleven training runs. At each generation when progress was made on the validation set, the nominal best network was also evaluated against the test set. The hatch marks at the bottom of the plot identify those generations. Test scores for those generations (only) are connected with straight lines to improve visibility. The plot is not strictly monotonic because progress on the validation set does not strictly imply progress on the test set

As described in Sect. 5.5.3, performance against the test set was also checked during training so that learning progress can be examined. A typical learning curve is shown in Fig. 5.7. The learning curve is familiar from many machine learning applications (though inverted because lower scores are better), with fast initial learning tapering off into slower steady progress. Experience shows that learning by neuroevolution on the *Legion II* problem appears to progress asymptotically. There is no stair-step pattern to suggest that the legions' two modes of behavior were learned sequentially; observations confirm that the legions begin chasing barbarians even before they have learned to garrison all the cities rigorously.

The behavior of a trained controller network can be evaluated qualitatively by observing real-time animations of game play. In every case that has been observed, trained legions begin the game with a general rush toward the cities, but within a few turns negotiate a division of labor so that some of the legions enter the cities or remain near them as garrisons while the others begin to chase down barbarian warbands in the countryside. The only time the cities are not garrisoned promptly is when their positioning allows two of them mask the third from the legions' low-resolution sensors. However, even in those cases the third city is garrisoned as soon as one of the roaming legions pursues a barbarian far enough to one side to have a clear view of the third city so that it can "notice" that it is ungarrisoned. A feel for these qualitative context-aware behaviors can be obtained by comparing end-of-game screenshots taken early and late during a training run, as shown in Fig. 5.8. An animation of the trained legions' behavior can be found at http://nn.cs.utexas.edu/keyword?ATA.

The legions' division of labor can can also be examined by the use of run-time metrics. The test program was instrumented to record, after each legion's move, how

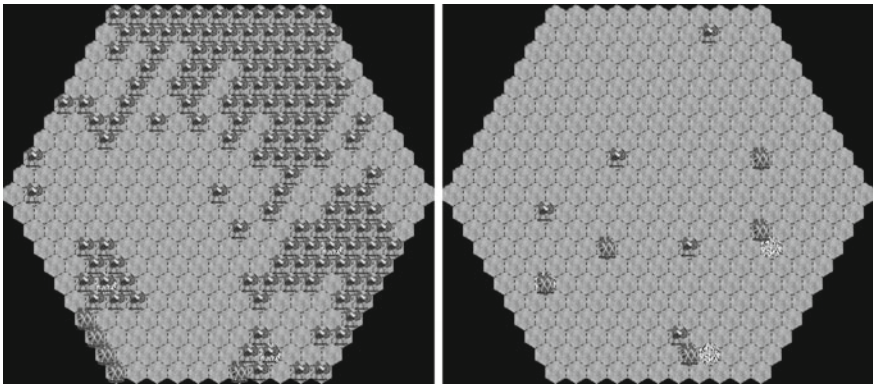

Fig. 5.8 Two end-of-game screenshots show the legions' performance before and after training. *Left*: Before training the legions move haphazardly, drift to an edge of the map, or sit idle throughout the game, thereby failing to garrison the cities and allowing large concentrations of barbarians to accumulate in the countryside. *Right*: After training the legions have learned to split their behavior so that three defend the three cities while the other two move to destroy most of the barbarians pillaging the countryside. The desired adaptive behavior has been induced in the team

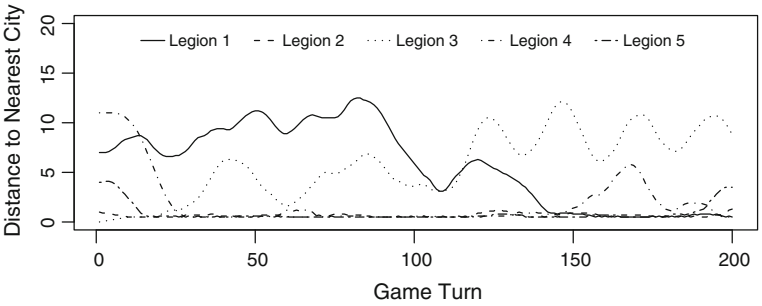

Fig. 5.9 The plot shows the distance from each legion to its nearest city over the course of a single game. The five legions start at random distances from the cities, but once some legion has had time to reach each of the three cities the team settles into an organizational split of three garrisons and two rovers. Note that the garrisons' average distance from their respective cities is not 0.0, because whenever there is only a single adjacent warband it is reasonably safe to exit the city long enough to eliminate it. The legions sometimes swap roles when a rover approaches a garrisoned city, e.g. Legions #3 and #4 just before turn 25. (The numbering of the legions is arbitrary; they are identical except for the happenstance of their starting positions. The lines have been smoothed by plotting the average of the values measured over the previous ten turns.)

far away it was from the nearest city. The result for allowing the median performer among the eleven trained networks to play the first game in the test set is shown in Fig. 5.9. The plot clearly shows that after a brief period of re-deploying from their random starting positions three of the legions remain very near the cities at all times while two others rove freely. The rovers do approach the cities occasionally, since that is where the barbarians primarily gather, but for most of the game they remain some distance away.

When a rover does approach a city there is sometimes a role swap with the current garrison, but the 3:2 split is maintained even after such swaps. However, the legions show surprisingly persistent long-term behavior for memoryless agents: the plot shows that Legion #1 acts as a rover for almost 3/4 of the game, and Legion #3, after starting as a garrison and then swapping roles with a rover, spends the final 7/8 of the game in that new role.

5.6.2 Run-Time Readaptation

The training games were parameterized to require the legions to organize a division of labor, and they successfully learned to do that. However, the motivation for the ATA multi-agent architecture in Sect. 5.1 calls for teams that can reorganize whenever a change in circumstances requires it. For example, if the pumper robot in the motivating example breaks down, one of the other robots should take over the task so that the team will not fail entirely. The legions in the *Legion II* game should also be able to reorganize at need.

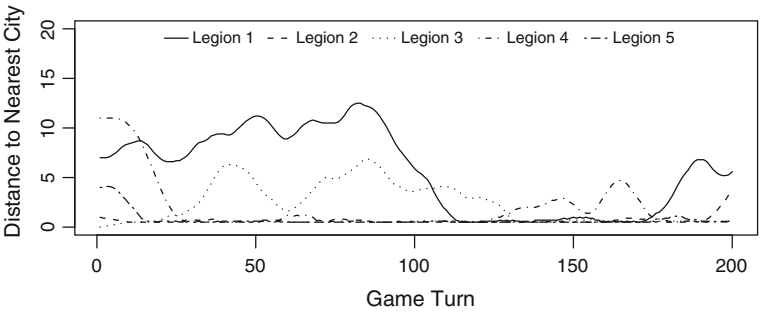

Fig. 5.10 In a second test, play follows the pattern of the previous test until turn 100, when a fourth city is added to the map. The team is forced to re-organize its division of labor so that there are now four garrisons and only a single rover. The role swaps continue, but they always leave four legions hovering near the cities. The typically lower average distance from the rover to its nearest city after the city is added is an artifact of the increased density of cities on the map. Legion #2 erroneously abandons its city near the end of the game; see the text for the explanation

That necessary ability was examined by modifying the test program to support the addition or removal of a city at the mid-point of the test games. When a city is added, the legions should reorganize into a team of four garrisons and one rover, or when a city is removed they should reorganize into a team of two garrisons and three rovers.

The result of adding a city is shown in Fig. 5.10. The plot again shows the median-performing controller's behavior on the first game in the test set. Since the legions' behavior is deterministic and the game's stochastic decisions are repeated, as described in Sect. 5.5.1, the game follows its original course exactly, up until the city is added at the mid-point of the game. Thereafter the team can no longer afford to have two rovers, and the plot shows the resulting reorganization. There are still role swaps in the second half of the game, but the swaps now always maintain four garrisons and a single rover.

The average distance from the rover to the cities is lower in the second half of the game. That is primarily an artifact of having more cities on the small map: regions that were once distant from all the cities no longer are, so even without any change in behavior the rover is expected to be nearer some city than before, on average. A second cause is an indirect result of the change in the team's organization. With only one rover in the field, the legions are not able to eliminate the barbarians as quickly as before, so during the second half of the game the concentration of barbarians on the map builds up to a higher level than previously. Since they tend to crowd around the cities and the roving legions tend to chase down the barbarians wherever they mass, the roving legion now has more reason to operate close to the cities.

The plot shows Legion #2 vacating the city it was garrisoning right at the end of the game. That is also an artifact of the increased density of the barbarians on the map. In ordinary play the trained legions are able to maintain a dynamic equilibrium between the rate of influx of the barbarians and the rate they are eliminated; the denser the barbarians are on the map, the easier it is for the rovers to catch some

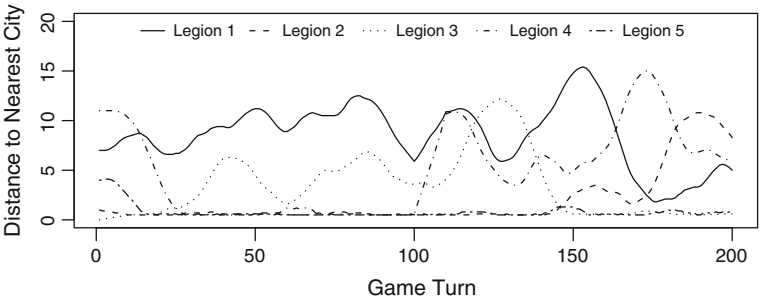

Fig. 5.11 In a third test, the mid-game reorganization experiment is repeated, except this time a city is *removed* from the map. That city had been garrisoned by Legion #4, which now finds itself far from the nearest city, and immediately adopts the role of a rover. (The steep ramp-up of its nearest-city distance on the plot is an artifact of the smoothing; the change is actually instantaneous when the city is suddenly removed from under the legion.) There is a role swap just before turn 150, but the team consistently keeps two legions very near the two remaining cities, with the other three roving at various distances

of them. However, when the add-city test causes one of the rovers to swap roles to garrison duty, that equilibrium can no longer be maintained by the single remaining rover, and the number of barbarians in play starts ramping up after the city has been added. Eventually their number oversaturates the legions' sensors – they have not seen such densities since early during training – and the legions begin to behave erratically. However, until their sensory input diverges quite far from what they were trained for, the legions are seen to exhibit the desired behavior.

The result of removing a city at the mid-point of a game is shown in Fig. 5.11. The play proceeds as before, until the city is removed at turn 100. At that point the legion formerly garrisoning that city finds itself far away from any, but it adopts the roving behavior rather than sitting idle or trying to crowd into one of the other cities, and it maintains that behavior for the remainder of the game. There is a role swap between two of the other legions later, but the team is always left with two legions hovering very near the cities on garrison duty, while the other three range over various distances in pursuit of the barbarians.

5.7 Discussion

The experiments show that the Adaptive Team of Agents is a feasible architecture for multi-agent systems, and that ATAs can be created by neuroevolutionary methods. The legions learned the desired variety of behavior, and the ability to organize a division of labor by individually adopting an appropriate choice of behaviors. They also learned to swap roles without disrupting the required organization of the team, both in the ordinary course of events and in response to a change in the scope of their task.

Such capabilities are essential for the agents embedded in many types of game or simulator. Games often provide multiple agents of some generic type – e.g. the settler type in the *Civilization* game – which must as individuals pursue differing activities that contribute to the success of the team rather than the individual. And those agents must be adaptive in their choice of activities, taking account of the game state, including the choices being made by their peers. Yet the scripted behavior of agents in commercial and open source games commonly fail at that requirement, making decisions that appear to take little account of context. For example, in strategy games, even when the agents are not autonomous and a higher-level AI is able to manipulate them according to some plan, individual agents are frequently observed to make piecemeal attacks that are suicidal due to a lack of supporting actions by their peers. To a human observer, such agents simply do not seem very intelligent. Appropriate computational intelligence methods should be able to take game intelligence beyond the brittleness, inflexibility, and narrowness of scripted activity, and for many games or simulators the context-awareness and adaptivity of the agents in an ATA will be a necessary part of any successful solution.

In the field of evolutionary robotics, Floreano et al. also examined homogeneous teams controlled by ANNs evolved by team selection, in a study of hypotheses for explaining biological altruism [8]. Altruism does not play an explicit role in *Legion II*, but their study found that a homogeneous team evolved by team selection performed better than three other architectures examined, producing robust altruistic behavior in the process. Altruism, when appropriate, is an important facet of trusted autonomy in multi-agent environments, and can contribute to the appearance of intelligent behavior as well.

It is interesting to note that the necessary adaptivity for our ATA was obtained using a simple feed-forward network for the legions' controllers. We know that artificial neural networks are powerful computing devices (see e.g. [7, 25]), and that genetic algorithms are able to train them to sophisticated behaviors (e.g. [1, 3, 15, 26, 27, 29]). To a first approximation it may be concluded that the *Legion II* controllers have been trained to partition the game's state space, as seen from an egocentric point of view, into two classes, and to choose a behavior on the basis of which class the current state observation falls in to. However, what they actually choose is one of seven atomic moves, none of which can be uniquely associated with either of the two behavior classes.

For an agent to pursue a coherent higher-level behavior across many game turns – i.e., to give an appearance of intent-driven behavior – would seem to require access to some internal state, i.e. an ability to "remember" what it is doing. Conjecturally, the *Legion II* agents have learned a workaround whereby they effectively store their internal state in the external environment. I.e., in addition to whatever else they learn during training, they learn a mapping from their egocentric view of the environment to a virtual representation of whatever internal state information is necessary for "remembering" what they are doing. The flow of information is in fact recurrent: the fact that the agents move within their environment causes a transformation of their next view of the environment. In an otherwise static environment those transformations would be deterministic; the presence of other agents in the *Legion II*

environment makes them somewhat noisy. However, the potential noisiness in the state transitions may be greatly reduced if the agents also learn an implicit model of the other agents' behaviors. If they can predict how those other agents will act, they can learn to predict what effect their choice of actions will have on their next snapshot view of their world with high accuracy. Thus it is possible, in principle, for the pairing of a sufficiently powerful computational device with a sufficiently powerful learning mechanism to learn to use an egocentric view of the external state as if it were an internal state, for systems such as the *Legion II* game. Future work must pursue this concept to determine what the limits of such a mechanism are. Does a feed-forward network embedded in an environment that it can manipulate become as powerful as a Turing machine?

The *ad hoc* use of plots of an *ad hoc* metric for detecting the legions' division of labor in Sect. 5.6 also reveals a need for developing methods of behavior analysis. When studying agents in visible environments such as games and simulators, behavior is paramount [3]. Meaningful behavioral metrics are essential, and it would be useful to have methods that are abstract enough to be portable across application domains, and sensitive enough to detect similarities or differences in behavior when a domain involves more subtlety than a switch between two discrete behaviors. Work in this area is already underway, and will be a major component of the study of *visibly intelligent behavior* in the future.

The *Legion II* ATA experiments also revealed a special challenge for the application of computational intelligence methods to agent behavior problems. The goal of a simulation as understood by a machine learning algorithm – e.g. minimizing pillage in the *Legion II* game – may be satisfied by some abstract optimization, with little or no regard for the appearance of details of the learned behavior. For example, the legions in the *Legion II* ATA experiment learned to switch between appropriate roles on the basis of context, but some of the details of their behavior are not satisfactory to an observer. The garrisons' learned behavior often produced "mindless" oscillations in and out of their cities when there were no barbarians nearby to threaten pillage, and such behavior would likely be the subject of ridicule if seen in the behavior of the agents in a commercial game. In principle such details of behavior can be addressed by careful specification of the goals of the training regimen, such as an evolutionary reward function that penalizes undesirable behavior, but for applications as complex as a commercial game it may be as difficult to specify an appropriate reward function as it has proven to be to write a script that covers all situations adequately. Therefore work is underway on suppressing such oddities of behavior and inducing other desirable traits that will make agents *look* intelligent to observers, rather than merely acting out some abstractly optimal solution to the problem they have been trained for. (See [3] for an extensive preliminary treatment.)

5.8 Conclusions

The *Adaptive Team of Agents* is a viable architecture for systems based on the methods of computational intelligence, and has immediate applications in domains such as games and simulators, where autonomous agents must show flexible diversity of behavior at both the individual and team level. Such flexibility, supported by fulfilled trust among the members of a team, is a critical component of trusted autonomy in multi-agent systems, and is a key aspect of the sort of visibly intelligent behavior that viewers expect from agents that model real or imagined creatures or groups in some simulated world. Neuroevolution in particular can create such flexibility, along with other desired characteristics of visibly intelligent behavior. More powerful neuroevolutionary methods continue to be developed, and it can be expected that further work in applying them to the rich challenges of modern videogames will produce results of both practical and scientific merit.

Acknowledgements This work was supported in part by NSF grant IIS-0083776, Texas Higher Education Coordinating Board grant ARP-003658-476-2001, and a fellowship from the Digital Media Collaboratory at the IC^2 Institute at the University of Texas at Austin. Most of the CPU time for the experiments was made possible by NSF grant EIA-0303609, for the Mastodon cluster at UT-Austin.

References

1. A. Agogino, K. Stanley, R. Miikkulainen, Online interactive neuro-evolution. Neural Process. Lett. **11**, 29–38 (2000)
2. T. Balch, *Behavioral Diversity in Learning Robot Teams*. Ph.D. thesis, Georgia Institute of Technology, 1998. Technical Report GIT-CC-98-25
3. B.D. Bryant, *Evolving Visibly Intelligent Behavior for Embedded Game Agents*. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2006
4. B.D. Bryant, R. Miikkulainen, Neuroevolution for adaptive teams, in *Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, vol. 3 (IEEE, Piscataway, NJ, 2003), pp. 2194–2201
5. B.D. Bryant, R. Miikkulainen, Evolving stochastic controller networks for intelligent game agents, in *Proceedings of the 2006 Congress on Evolutionary Computation (CEC 2006)* (IEEE, Piscataway, NJ, 2006), pp. 3752–3759
6. B.D. Bryant, R. Miikkulainen, Exploiting sensor symmetries in example-based training for intelligent agents, in *Proceeedings of the 2006 IEEE Symposium on Computational Intelligence and Games (CIG'06)*, ed. by S.J. Louis, G. Kendall (IEEE, Piscataway, NJ, 2006), pp. 90–97
7. G. Cybenko, Approximation by superpositions of a sigmoidal function. Math. Control Signals Syst. **2**(4), 303–314 (1989)
8. D. Floreano, S. Mitri, A. Perez-Uribe, L. Keller, Evolution of altruistic robots, in *Computational Intelligence: Research Frontiers: IEEE World Congress on Computational Intelligence*, WCCI 2008, Hong Kong, China, June 1–6, 2008, Plenary/Invited Lectures, ed. by Jacek M. Zurada, Gary G. Yen, Jun Wang (Springer, Berlin, 2008), pp. 232–248

9. F. Gomez, *Robust Non-Linear Control Through Neuroevolution*. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin, 2003
10. F. Gomez, R. Miikkulainen, 2-D pole-balancing with recurrent evolutionary networks, in *Proceedings of the International Conference on Artificial Neural Networks* (Springer, Berlin, 1998), pp. 425–430
11. F. Gomez, R. Miikkulainen, Solving non-Markovian control tasks with neuroevolution, in *Proceedings of the 16th International Joint Conference on Artificial Intelligence* (Morgan Kaufmann, San Francisco, CA, 1999), pp. 1356–1361
12. F. Gomez, R. Miikkulainen, Active guidance for a finless rocket using neuroevolution, in *Proceedings of the Genetic and Evolutionary Computation Conference* (Morgan Kaufmann, San Francisco, CA, 2003), pp. 2084–2095
13. T.D. Haynes, S. Sen, Co-adaptation in a team. Int. J. Comput. Intell. Organ. **1**, 231–233 (1997)
14. J.E. Laird, M. van Lent, Human-level AI's killer application: interactive computer games, in *Proceedings of the 17th National Conference on Artificial Intelligence* (AAAI Press, Menlo Park, CA, 2000)
15. S.M. Lucas, Cellz: a simple dynamic game for testing evolutionary algorithms, in *Proceeedings of the 2004 Congress on Evolutionary Computation (CEC 2004)* (IEEE, Piscataway, NJ, 2004), pp. 1007–1014
16. R. Miikkulainen, B.D. Bryant, R. Cornelius, I.V. Karpov, K.O. Stanley, C.H. Yong, Computational intelligence in games, in *Computational Intelligence: Principles and Practice*, Chap. 8, ed. by G.Y. Yen, D.B. Fogel (IEEE Computational Intelligence Society, Piscataway, NJ, 2006), pp. 155–191
17. D. Moriarty, R. Miikkulainen, *Learning sequential decision tasks*. Technical Report AI95-229, Department of Computer Sciences, The University of Texas at Austin, 1995
18. D.E. Moriarty, *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. Ph.D. thesis, Department of Computer Sciences, The University of Texas at Austin, 1997. Technical Report UT-AI97-257
19. D.E. Moriarty, R. Miikkulainen, Discovering complex Othello strategies through evolutionary neural networks. Connection Sci. **7**(3), 195–209 (1995)
20. D.E. Moriarty, R. Miikkulainen, Evolving obstacle avoidance behavior in a robot arm, in *From Animals to Animats 4: Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior*, ed. by P. Maes, M.J. Mataric, J.-A. Meyer, J. Pollack, S.W. Wilson (MIT Press, Cambridge, MA, 1996), pp. 468–475
21. M.A. Potter, K.A. De Jong, Evolving neural networks with collaborative species, in *Proceedings of the 1995 Summer Computer Simulation Conference* (1995)
22. R Development Core Team, *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria, 2004
23. N. Richards, D. Moriarty, P. McQuesten, R. Miikkulainen, Evolving neural networks to play Go, in *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA-97, East Lansing, MI)*, ed. by T. Bäck (Morgan Kaufmann, San Francisco, CA, 1997), pp. 768–775
24. J.D. Schaffer, D. Whitley, L.J. Eshelman, Combinations of genetic algorithms and neural networks: a survey of the state of the art, in *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks*, ed. by D. Whitley, J. Schaffer (IEEE Computer Society Press, Los Alamitos, CA, 1992), pp. 1–37
25. H.T. Siegelmann, E.D. Sontag, Analog computation via neural networks. Theor. Comput. Sci. **131**(2), 331–360 (1994)
26. K.O. Stanley, B.D. Bryant, R. Miikkulainen, Evolving adaptive neural networks with and without adaptive synapses, in *Proceeedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, vol. 4 (IEEE, Piscataway, NJ, 2003), pp. 2557–2564
27. K.O. Stanley, B.D. Bryant, R. Miikkulainen, Real-time neuroevolution in the NERO video game. IEEE Trans. Evol. Comput. **9**(6), 653–668 (2005)
28. X. Yao, Evolving artificial neural networks. Proc. IEEE **87**(9), 1423–1447 (1999)
29. C.H. Yong, R. Miikkulainen, *Cooperative coevolution of multi-agent systems*. Technical Report AI01-287, Department of Computer Sciences, The University of Texas at Austin, 2001