

Hierarchical Controller Learning in a First-Person Shooter

Niels van Hoorn, Julian Togelius and Jürgen Schmidhuber

Abstract—We describe the architecture of a hierarchical learning-based controller for bots in the First-Person Shooter (FPS) game *Unreal Tournament 2004*. The controller is inspired by the subsumption architecture commonly used in behaviour-based robotics. A behaviour selector decides which of three sub-controllers gets to control the bot at each time step. Each controller is implemented as a recurrent neural network, and trained with artificial evolution to perform respectively combat, exploration and path following. The behaviour selector is trained with a multiobjective evolutionary algorithm to achieve an effective balancing of the lower-level behaviours. We argue that FPS games provide good environments for studying the learning of complex behaviours, and that the methods proposed here can help developing interesting opponents for games.

Keywords: First-person shooters, FPS, evolutionary algorithms, neural networks, behaviour-based robotics, subsumption architecture, action selection

I. INTRODUCTION

First-person shooter games are three-dimensional combat simulation games which are viewed from a first-person perspective, and where the player is tasked with surviving in an adversarial environment through winning firefights with other agents. In many FPS games the player takes control of an armed soldier on a battlefield, which might simulate aspects of historical battles (e.g. the *Call of Duty* series) or science fiction scenarios (e.g. the *Halo* series), fighting against enemies such as other soldiers or monsters.

Playing an FPS well requires mastering a number of related but distinct skills. To begin with, there are the lower level perceptual and motor skills, such as quickly identifying that something moved on a part of the screen, identifying what it was (friend or foe?) and reacting appropriately (e.g. aiming at the moving object, or backing away). Intermediate level skills require simple planning and include deciding in what order to attack enemies when several are present in a room, selecting appropriate weapons for the current battle, and finding and moving to good positions from where the player can take aim at enemies without needing to worry about being attacked from behind. Higher level “cognitive” skills are concerned with creating a complex representation of the environment and include mapping the area the player is moving in, keeping track of the positions of health-packs, ammunition supplies and enemies, and planning where to explore and what resources to gather before particular battles.

In other words, playing an FPS well requires many of the capabilities that have traditionally been studied within computational and artificial intelligence (in team-based combat communication skills become relevant as well). FPS games

JT is with the IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen S, Denmark. NvH and JS are with IDSIA, Galleria 2, 6928 Manno-Lugano, Switzerland. Emails: {niels, julian, juergen}@idsia.ch



Fig. 1. A scene from Unreal Tournament 2004

are vastly cheaper, simpler to handle and faster to run than physical robots, but also more complex and demanding than the toy problems traditionally used in CI research. We argue that such games are good testbeds for research on learning or otherwise developing controllers that perform complex tasks, in the sense of being composed of several simpler tasks.

A. Game AI and CI in first-person shooters

All FPS games that feature computer controlled non-player characters (NPCs), also known as *bots*, come with some form of game AI. These bots are usually controlled by algorithms that, while sometimes very sophisticated and frequently very appropriate for their purpose (entertaining the human player of the game), do not include any form of learning and do not play the game under the same conditions as a human does. Traditionally, bots are controlled by finite state machines built out of a number of hard-coded rules that define reactions to particular stimuli and transitions to other states; recently, behaviour trees have started to replace finite state machines as the controller representation of choice. The higher-level navigation is usually done through the A^* path finding algorithm with predefined navigation points. Often, the algorithm controlling the bot has access to considerably more information than the human player has (e.g. can see through walls), and (equally important) this information is usually represented from a third-person perspective. e.g. as navigation points in a map instead of first-person sensors.

The above description is rather coarse and there are many important exceptions (e.g. the use of probabilistic techniques by enemies searching for the player in *Halo 2* [1]). However, current commercial game AI has little to do with current academic research in the AI and CI communities. On the other hand, such games have been used for academic research.

Previous applications of CI to FPS games can be divided in two dimensions. The first being if the controller has learned to perform only a partial task of the gameplay or the full one, and the second dimension makes the distinction if hardcoded elements are used in the controller or the controller only used primitive (player-centered) inputs and actions (e.g. moving, turning and shooting).

We plan to include a full survey of previous applications in a forthcoming paper, here we are limited to references due to spatial constraints. For learning partial gameplay, Overholtzer and Levy [2] used hardcoded elements, while Kadlec et al. [3], Karpov et al. [4], Parker and Bryant [5], Priesterjahn [6] and Thurau et al. [7] addressed different parts of the controller using primitive actions. Cole et al. [8] learned controllers for the full gameplay task using some hardcoded elements, like Small and Congdon [9] and Westra [10], while McPartland and Gallagher [11] used only primitive actions, though in a purpose-built FPS.

As far as we are aware, all attempts at learning FPS bot behaviour that have resulted in human-level playing performance have built on heavily preprocessed environment representations which have little in common with human visual input and with sensors that could be fitted on a robot. Most of the attempts above also treat the bot controller as a monolithic (non-differentiated) system, which is learnt at the same time during the execution of a single task.

B. Learning hierarchical controllers

Behaviour-based robotics has been a dominant paradigm in robotics for the last two decades [12]. In this paradigm, the robot is controlled by a *layered* or *hierarchical* control system. Each layer performs a well-defined subtask (e.g. in the case of a traffic rule abiding robot: keeping a desired speed, staying on the road, avoiding pedestrians, stopping in front of red lights), and the breakdown of the robot task into subtasks is performed so as to allow each layer to be as simple as possible, and thus respond as quickly as possible to changes in the robot's environment. Many different types of behaviour-based architectures exist, and the dominance relations between layers vary considerably. In most cases, however, the lower layers implement more "primitive" behaviours and the higher layers balance or organize the contributions of the lower layers.

Some behaviour-based architectures are completely hardcoded, others incorporate learning as part of some layers. A few attempts have been made to learn the functionality of the layers themselves. Togelius proposed the *layered evolution* method, where each layer is represented as a neural network and evolved separately, starting from the lowest layers and keeping the weights of lower (already evolved) networks frozen while evolving higher layers [13]. Each time a higher layer is added, the complexity of the task is incremented [14]. Thompson and Levine recently used a similar method to develop a layered controller for the *EvoTanks* game [15].

C. Using FPS games for evolutionary robotics research

Evolutionary robotics is concerned with evolving robot controllers, usually represented as neural networks, that allow robots to solve specific predefined tasks [16]. While seeming to hold great promise initially, this research direction has seen only limited progress in evolving solutions to complex control problems, especially those that require sequential execution of diverse behaviour and the handling of multidimensional environment representations. We venture that this is partly because of the difficulty of using physical robots for evolutionary experiments. Modern FPS games provide experimental environments that have several advantages over robots used in the real world.

Experiments in an FPS game require no specific hardware, and can be sped up and/or parallelized through distribution over several cores in a computer or several computers in a cluster. Modern games provide advanced physics, elaborate and varying environments with predefined tasks (requiring an array of diverse cognitive capacities, as discussed above) complete with accurate reinforcements (i.e. score). Sophisticated graphics allow for high-dimensional simulated sensing. As a commercial computer game is typically the result of a hundred or more people working for a year or more, and has been tested by many thousands of players, such games are usually also bug-tested and detail-rich to an extent not possible in typical robotics simulators.

Another reason for the apparent partial stagnation of evolutionary robotics could be that not much effort has been spent on learning hierarchical architectures with task-switching (but see exceptions discussed above), something which probably is the key to solving complex tasks.

In this paper, we try to address both of these concerns, evolving a hierarchical controller for an agent in a modern FPS game.

D. Aims and scope of this paper

This paper describes the architecture of a hierarchical controller, where each layer is based on a neural network and trained separately with an evolutionary algorithm, for bots in a modern FPS game. The aim of this paper is to show that such an architecture and development method can result in relatively high-performing FPS bots, using only low-level first person environment representations as inputs. We emphasize that we are *not* trying to outperform the best hard-coded bots that make use of environment information that is hidden and/or represented in a third-person format. Indeed, manually developing a bot that outperforms most human players would be relatively easy using the high-level behavioural primitives available in the game, such as automatic aiming, but this would not be very interesting. Instead we are seeing the FPS game as an environment in which to perform evolutionary robotics-style experiments to demonstrate the power of our particular controller architecture given a realistically restricted environment representation.

The secondary aims of the paper are to elucidate the most successful design choices for hierarchical agents and to

compare the performance of hierarchical agents and agents based on undifferentiated (monolithic) neural networks.

In the following we first describe the methods used: the FPS game, sensor representation and action space, the neural networks and evolutionary algorithms used and the hierarchical architecture. We then describe the training of the individual sub-controllers and of the behaviour selector.

II. METHODS

A. Unreal Tournament 2004 and Pogamut

Unreal Tournament 2004 (in the following frequently abbreviated *UT2004* or just *UT*) is a popular commercial FPS game (see figure 1) which is particularly noteworthy for its multiplayer features, and for the fact that the underlying game engine has been reused in a number of other commercial computer games. Part of the long-running *Unreal* series of FPS games, it's a mature and thoroughly tested game, ensuring that it contains few of the type of bugs that are often discovered and exploited by evolutionary algorithms [17].

For this game, there has been a concerted effort to make it possible to control all the agents in the game from external applications. On the server side, *GameBots* [18] is a modification of the Unreal environment written in UnrealScript that makes it possible to get information about a bot and its environment through a TCP/IP connection. This information can be used to send controlling commands back to the bot, thus completing the sensorimotor loop.

Originally developed at the University of Southern California and continued by a team from the Technical University of Prague, *Pogamut* [19] provides a wrapper for the GameBots environment in the form of a java package, complete with a rich API containing high and low-level functions to access the sensors and affect the controls of UT2004 agent. Pogamut has been used several times as a learning environment for AI research, most notably by Kadlec [3], who also created a system to distribute experiments over a cluster.

In UT lengths and distances are measured in UT units¹. One UT unit roughly maps to 0.75 inches, so 1 meter relates to roughly 52.5 units. Unless otherwise mentioned, all measurements concerning distances are given in UT units.

B. Sensing

The agent is supplied with a suite of sensors designed to operate from a first-person perspective, and not provide the agent with any information that a human player would not have had access to. The following sensors are used:

- **Ray-tracing wall sensors** The agent is equipped with 12 sensors to detect the walls around it. Each wall sensor has an angle, relative to the direction the agent is facing. The wall sensor returns a value between 0 and 1, proportional to how far away a wall is encountered in that direction (similar to the laser range finding sensors used in robotics). If no wall is encountered within 1000 UT units, the sensor returns the value 1; if the agent is

¹See <http://wiki.beyondunreal.com/Legacy:Unreal.Unit> for more info

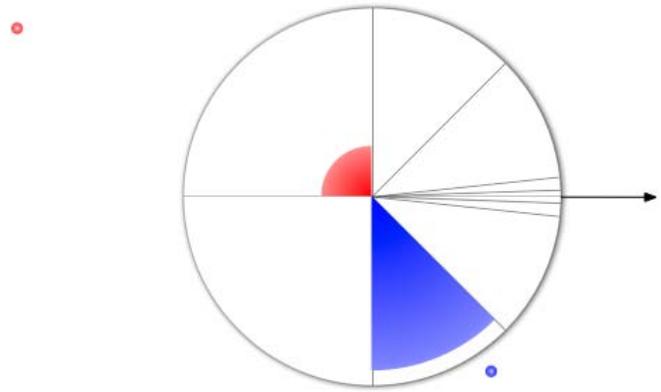


Fig. 2. An abstract representation of the pie-slice enemy sensor. The arrow is indicating the direction of the agent is facing, and the red and blue dots symbolize enemies that get mapped onto their respective slices.

standing next to a wall in the probed direction it returns a value close to 0.

- **Pie-slice enemy sensors** The enemy sensors work similarly to the wall sensors, with the exception that they each cover a “pie-slice” (the area defined by a circle segment centered on the agent and with a given angle) of the environment. This is because enemy agents are smaller than wall segments, and thus harder to detect through ray-tracing. The closer the enemy is, the higher the value of the slice. The sensors are divided into 12 slices of unequal size; see figure 2. The front slices cover an angle of $\pi/128$ radians, where the sensor at the back covers $\pi/2$ radians. This gives the agent a high precision in the front with a modest total number of inputs.
- **Direction to next way point** To facilitate path following the bot is given the relative angle and distance to the next waypoint of the path it needs to travel. The path is calculated by UT and the bot always goes to the nearest known item of the specified kind. When a new item is discovered it is added to the list of known items.
- **Health** The current health level divided by 100 (to normalize the sensor input; health can reach 199).
- **Being damaged** 1 if the bot is currently taking damage, 0 otherwise.

C. Actuating

There are several actions the GameBots interface allows to be sent to the bot. Unfortunately these actions are better suited to more traditional scripting than robot-like control of the bot. For example, it is possible to let the bot walk to a specific location given by euclidean coordinates and simultaneously look another coordinate in space using a *Strafe* command, but Pogamut does not offer the possibility to steer the bot by more primitive “turn” and “move” actions; if “turn” and “move” were both sent to the bot, the latter command would cancel out the former. Because of these restrictions we implemented robot-like moving and turning with the use of the Strafe command. However, as

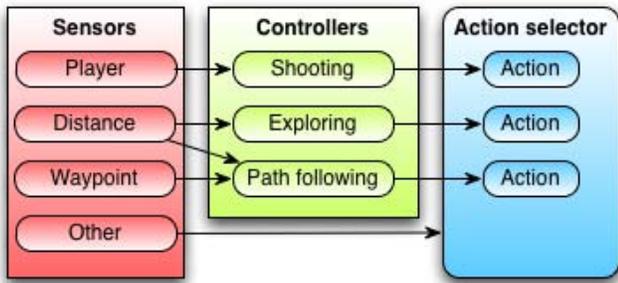


Fig. 3. Overview of the hierarchical structure

the directions of the bot are calculated relative to its current position and the bot might be at a different position when the action is performed, this method is slightly inaccurate.

Each time step, the controller outputs values for the following actions:

- **Moving** is defined within a range of $[-2, 2]$ where a negative value means moving backwards, a positive value forward, and normal speed is 1.
- **Turning** values range between $[-\pi, \pi]$ and are interpreted as the number of radians the bot should turn.
- **Shooting** can be true or false.

Notice that there is no way for the bot to look up or down; instead we modified maps so as to only have one floor. Relatedly, the bot always crouches when shooting, so as to be able to hit crouching opponents.

D. Hierarchical architecture

Based on experience from late-night gaming sessions, we decomposed the skill necessary for playing UT2004 well (in deathmatch mode) into the following sub-skills:

- *Shooting* is arguably the most important skill for playing any FPS. The challenge here is to detect when enemies are nearby, select which enemy to attack and, for a given weapon, inflict as much damage as possible within a given time. This involves aiming well, taking the characteristics of the weapon into account, and repositioning the bot relative to its target.
- *Exploration* is a crucial skill for any environment except a small room with complete visibility. The challenge is to chart out as much as possible of the environment in as short time as possible, finding any health packs, ammo stashes and enemies.
- *Path-following* becomes important in environments where vital resources such as health and ammunition regularly run low but can be replenished at locations scattered around the environment. The challenge is to get to a given location (e.g. a health pack) as quickly as possible, preferably moving in such a way as to minimize the risk of getting shot.

Behaviour selection or *action selection* means switching between the three sub-skills listed above. The challenge here is to choose when to shoot, when to explore, and when to

run for the next health pack, depending on the agent's current resource levels, immediate environment, and history.

The architecture used to learn the behavior is depicted in Figure 3. The sensors displayed on the left side are the sensors described in section II-B. These sensors are fed into 3 different controllers that are trained in separate experiments with separate fitness functions. Each controller is intended to implement one of the key sub-skills discussed above, and the fitness functions used to train that controller are intended to measure its proficiency at the particular skill. When these controllers reach good fitness on their separate tasks, they are frozen (the weights of the neural network are not allowed to change further). Then the best individuals for each task are selected and used in a different experiment where the action selector is trained.

E. Neural networks and evolutionary algorithms

All sub-controllers are implemented as recurrent neural networks and trained with evolutionary algorithms.

Some controllers were implemented as Simple Recurrent Networks (SRN), also known as an Elman Networks, with *tanh* activation functions. An SRN is the same as a standard Multi-Layer Perceptron, except that each neuron in the hidden layer also has inputs from all neurons of the hidden layer of *the previous time step* (the last time values were propagated through the network) [20]. Other controllers were implemented as Long-Short Term Memory (LSTM) networks. LSTM is an architecture especially designed to capture long-term time dependencies that has previously exhibited world-class performance on sequence learning tasks such as speech recognition [21].

We used two different evolutionary algorithms. Some controllers were evolved with single objective each; in these experiments, a standard $\mu + \lambda$ Evolution Strategy (ES) was used with $\mu = \lambda = 25$. The ES was stopped as soon as the fitness of the best individual had not improved for 20 generations. Other controllers were evolved with multi-objective evolution, and here the NSGA-II [22] algorithm was used, being one of the most widely used multiobjective evolutionary algorithms with a reputation for robustness, with a population size of 100 for 100 generations.

The weights of the SRNs were initially set to random numbers drawn from $[-1, 1]$ and mutation was performed by adding a normally distributed value $X \sim N(0, 0.1)$. The weights of the LSTM networks were initialized with random numbers from $[-0.1, 0.1]$. During mutation a number drawn from a Cauchy distribution with location $x_0 = 0$ and scale $\gamma = 0.01$ is added to each weight. Because the Cauchy distribution has a so called 'fat tail' compared to the normal distribution, it can be advantageous to use it to escape local minima. No crossover was used in any of the experiments. All networks were fed a constant bias input in addition to the sensory inputs described in section II-B.

F. Maps

For the experiments we used three different maps to test the bots' performance. We used existing UT maps, but

modified them to be able to handle our simplifications of the full UT game. We decided to remove armor and only use a ShockRifle² for our experiments, to eliminate the need for item and weapon selection. We also only used one floor in each level to reduce input and control dimensionality.

- **DM-TrainingDay-Shock** is a modified version of the map DM-TrainingDay, an 8-shaped map that is shipped with Unreal Tournament. We removed the adrenaline and replaced all weapons and ammo with ShockRifles and ShockRifle ammo respectively. All removed items were replaced with path nodes, to keep the graph of nodes identical to the original map.
- **DM-1on1-Trite-Floorlevel** is a modified version of the map DM-1on1-Trite that is shipped with Unreal Tournament. This map contains the same modifications as DM-TrainingDay-Shock plus some others. The ramps and elevators going from the ground floor to the upper floors are removed and only four spawning positions are placed on the ground floor, thus making the upper floors inaccessible and reducing the map to a single floor. Additionally, the Armor shield on the ground floor was removed and replaced by a Big Keg O' Health and the 4 Health Vials surrounding it were removed.
- **DM-Bigroom** is a map we created ourself consisting of a single square room with a side of 1000 UT units.

III. EXPERIMENTS

In this section we first describe our attempts at evolving each of the three sub-controllers independently, and how the action selection controller was evolved on top of the already evolved and frozen sub-controllers. We then compare those results with the evolution of a monolithic controller evolved under the same conditions as the action selector.

A. Evolving exploration

For these experiments a bot was spawned in DM-TrainingDay-Shock or DM-1on1-Trite-Floorlevel and was given the 12 distance sensors and a bias of 1 as input. The SRN contained 8 hidden nodes and 2 outputs, that mapped to move and turn actions. Its task was to explore the map by visiting as many pathnodes of the map as possible in 30 seconds, after which the experiment was terminated. A pathnode was considered visited if the bot at some timestep had a distance of 100 or less to the pathnode. When the agent visits a node, the node value is set to 1, but it slowly decays every timestep. The fitness is the average value of all pathnodes at the end of the experiment. This was used in a weighted sum with a value proportional to the negative number of wall collisions. A formal representation of this fitness function is given in equation (1).

$$F_{\text{explore}} = 0.8 \cdot \frac{\sum_{i=0}^k n_i \cdot f^{(T-t_{n_i})}}{k} + 0.2 \cdot e^{(-w/5)} \quad (1)$$

²For those not familiar with UT2004 and its terminology, <http://liandri.beyondunreal.com/Deathmatch> contains links to the different items and weapons used in the game.

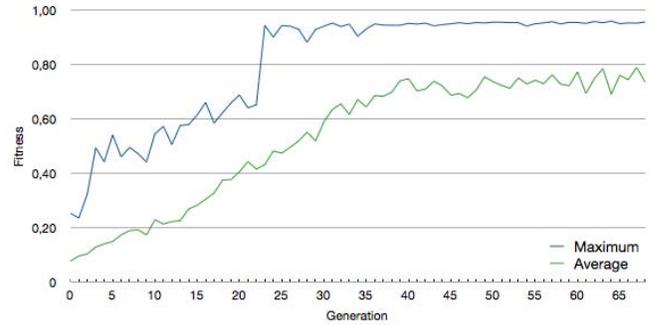


Fig. 4. Fitness of exploring in DM-TraingDay-Shock

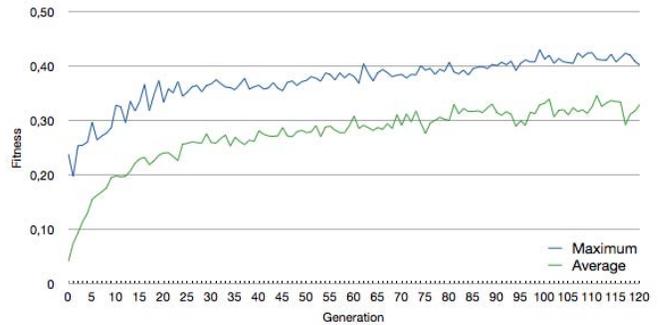


Fig. 5. Fitness of exploring in DM-1on1-Trite-Floorlevel

Where k is the number of pathnodes, n_i is 1 if node n_i is visited and 0 otherwise, f is the forget factor where $0 \leq f \leq 1$, T is the number of timesteps of the experiment and t_{n_i} is the timestep when the node n_i was visited last. Finally, w is the number of times the agent hit against a wall.

The exact value of the forget factor turned out to be unexpectedly important. We tried the same experiment with values $f = 0.99$ and $f = 0.999$, and with the former the decay of node values was too high so the agent only evolved a local exploring behaviour; we therefore used the latter value.

The result of exploration in DM-TrainingDay-Shock is shown in figure 4. The fitness increases a lot in the first generations and makes a final jump in generation 23, after which the fitness still increases, but only moderately. This is probably caused by the high regularity of the 8-shaped map. First the agent learns to cover only one loop of the 8, but suddenly it learns to explore both loops, and the found solution is close to optimal and doesn't improve much more.

In the other map, DM-1on1-Trite-Floorlevel, fitness increased much slower and more gradually, as can be seen in figure 5. This is probably because the latter map is bigger and structured with more rooms and passages between them.

Notice that the fitness reached in DM-1on1-Trite-Floorlevel is lower than in DM-TrainingDay-Shock. This is mainly because around 32% of the pathnodes is at some higher floor in the map, and thus unreachable by the bot (as described in section II-F).

The best evolved controllers explore the complete map

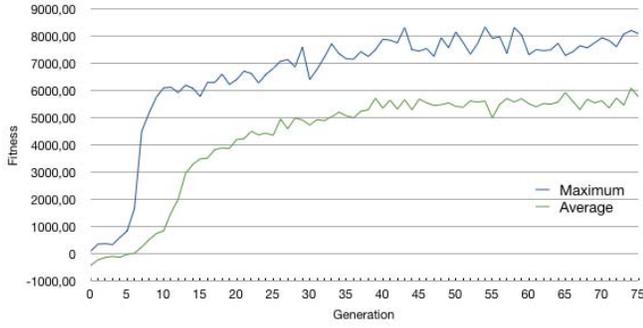


Fig. 6. Fitness of path following in DM-TraingDay-Shock

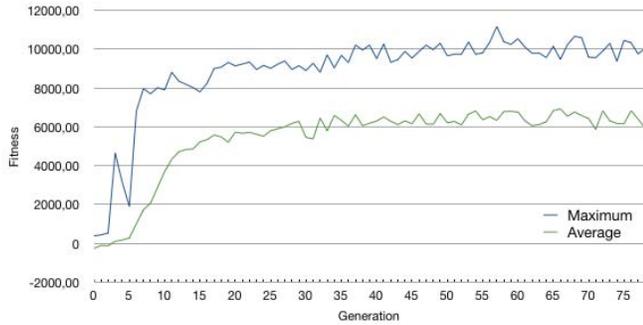


Fig. 7. Fitness of path following in DM-1on1-Trite-Floorlevel

they learned on, while only seldom running into walls. The behaviour is a rather non-intuitive pattern of the bot “feeling” its way around a map, a pattern that would be unlikely to be programmed by a human, especially given these inputs.

B. Evolving path-following

The path following controller gets 12 distance sensors to the walls as well as the distance and the angle to the next path node and a bias of 1 as inputs. The SRN used had 8 hidden nodes and 2 outputs, that map to move and turn actions. Paths are created by selecting the nearest item in the map that has not been visited for 27.5 seconds (the respawn time of most items in UT) and let UT find a path to that item. When the item is reached (i.e. the bot is within distance of 60 or less), a new nearest item is selected. In the small maps we used, this creates an infinite path that visits all items. The fitness of the agent is given by the length of the path travelled by the agent, plus the distance already travelled in the direction of the next node on the path; see equation (2).

$$F_{\text{path}} = \sum_{i=1}^{k-1} d(n_i, n_{i+1}) - d(l, n_k) \quad (2)$$

Where k is the number nodes in the path, the k th one being the pathnode that is next to be visited, $d(n, n)$ is the metric distance function, n_i is the i th visited node by the agent (n_1 being the node starting node of the agent) and l is the location of the agent when the experiment was terminated.

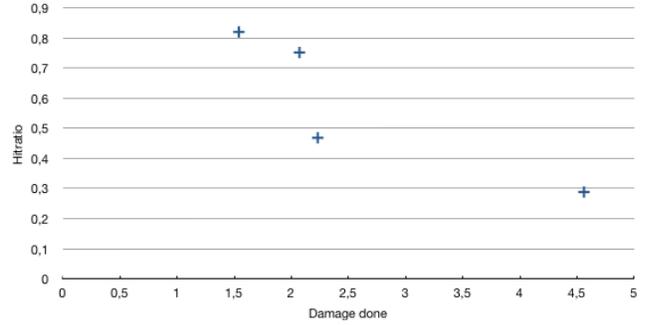


Fig. 8. Pareto front of evolving shooting

The results of the path following evolution is shown in figures 6 and 7. As can be seen from these graphs, the agent learns rather quickly to follow the path in both maps, and does not improve a lot after that.

C. Evolving shooting

The shooting experiments were performed in DM-Bigroom. The bot was given a ShockRifle with infinite ammo and placed at a random position in the map. As inputs it received a bias and the values of the 20 pie-slice enemy sensors. The SRN had 14 hidden nodes and 3 outputs, that map to move, turn and shoot actions.

In this experiment the opponent would never move or shoot at the agent. The goal of the agent was to kill as many opponents as possible within 30 seconds. Whenever an opponent was killed, it immediately respawned at a random position. Because we found it hard to find a good balance between the number of opponents killed and the accuracy of the shooting in a single fitness function, we evolved the controller multiobjectively. One fitness function measured the amount of damage the agent caused, while the other measured the proportion of bullets that hit their target. These fitness functions are given in equations (3) and (4).

$$F_{\text{damage}} = \text{kills} + \frac{\text{damage}}{100} \quad (3)$$

$$F_{\text{hitratio}} = \frac{\text{hitShots}}{\text{firedShots}} \quad (4)$$

Where kills is the number of times the agent killed an opponent, damage is the damage the opponent has received since his last respawn, hitShots is the number of shots that hit their target and firedShots the number of shots fired.

The results of this experiment is shown in figure 8. The pareto front only consists of four points, probably because of the noisiness of the fitness function. As we evaluate each individual only three times, the individuals that achieve good values in all runs dominate the other individuals. When looking at the behaviour, there are two distinguishable groups in the pareto front. The three controllers in the top left of the graph run around in circles until they encounter the opponent, place a well aimed shot and run another circle. The remaining controller turns around on the spot until it sees the

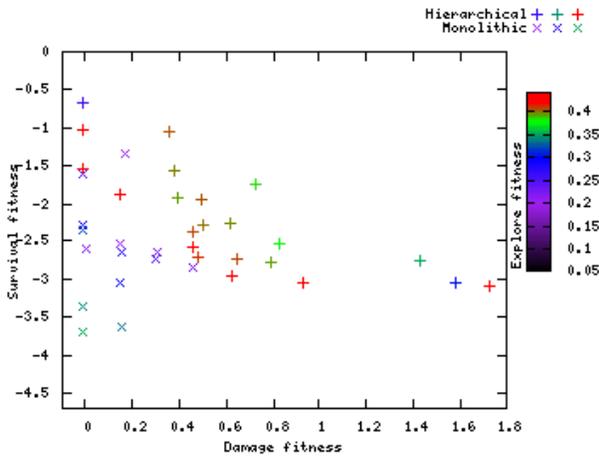


Fig. 9. Pareto fronts of hierarchical (+) and monolithic (x) controllers.

opponent. It then walks towards the opponent while aiming and shooting. The behaviour of the last controller is in our eyes much more human-like. It was also the most proficient at the main task: killing the opponent. We therefore chose this controller for use in the behaviour selection experiment.

D. Evolving behaviour selection

As noted in section II-D the hierarchical controller was evolved in stages. First the sub-controllers were evolved on their separate tasks (described above), and then the sub-controllers were frozen and the action selector was evolved.

This action selector is implemented as an LSTM network that receives 4 inputs: a bias, the health level of the agent, the sum of the player pie-slice sensors and whether the agent is currently taking damage. The size of the hidden layer was 5 and the 3 outputs represented the three different behaviours. Each time step the action produced by the controller corresponding to the highest output is used. Only one action is sent to the agent at any one time.

The experiments were performed in DM-1on1-Trite-Floorlevel, where the agent played against the lowest level UT bot. Multiobjective evolution was used with three objectives: to cause as much damage as possible, to take as little damage as possible and to explore the environment. These fitness functions are given by equations (3), (5) and (1) respectively. The first two fitness functions are a continuous version of the score in UT. The third objective was added to increase the diversity in the population; initial experiments showed that evolution with only two objectives got stuck in the local optimum of stationary bots. Standing still seems to be a good strategy when playing against a single opponent.

$$F_{\text{survival}} = -\text{deaths} - \left(1 - \frac{\text{health}}{100}\right) \quad (5)$$

Where *deaths* is the number of times the agent died and *health* is the health of the agent at the end of the experiment.

The results of evolving the behaviour selection module are shown with + in figure 9. In this graph fitness values

of the three objectives are shown. There is a tradeoff for the agent between doing more damage to his opponent and taking less damage itself. And although the variation in exploration fitness is small, there is an apparent tradeoff between exploration and the other objectives.

Looking at the behaviour of the evolved controllers, some show quite natural playing behaviour: running around the items in the map and engaging in firefights with the opponent, but the discrete switching between sub-controllers is clearly visible in the emerged behaviour. The reason for this is that these subcontrollers have distinctively different behaviour on both micro and macro scales.

Although the agent is able to win some firefights against the UT bot, it often runs recklessly towards the opponent without avoiding incoming fire. This is understandable, as the sub-controllers were not trained against bots that shot back.

The controllers in the upper-left part of the graph in figure 9 survive well through avoiding the opponent and rarely returning fire. Instead, they run around the map and often ignore the opponent completely. Although this cowardly is understandable and works quite well, it is not the kind of behaviour we were aiming for with our experiments.

E. Combined evolution from scratch

To show the benefits of our hierarchical architecture we ran an experiment with one single SRN. The inputs were all the sensors described in section (II-B), the network had a hidden layer of 15 neurons and 3 outputs described in II-C. Setup of the test and the fitness functions used were the same as described in section III-D. Our results so far are shown with x in figure 9. Although we would like to repeat the experiments and give them more evaluation time to make this statement stronger, it can be seen that all the controllers implemented as monolithic networks are dominated by hierarchical controllers. The evolved controllers show a pretty simple behaviour of turning or circling. This behaviour does not seem to change much in the presence of an opponent. The bots do not explore the map, usually staying in the room as they spawned in.

IV. DISCUSSION

One can look at the proposed architecture and the presented results in this paper from the computational intelligence perspective and the games perspective.

From the computational intelligence perspective, we have provided another demonstration of the power of a relatively under-explored technique for creating controllers for embodied agents: representing the components of a hierarchical controller architecture as neural networks and evolving them separately. In our opinion, the task solved is at least as complex as any that has been successfully solved in evolutionary robotics; consider the number of different skills needed (and the need to coordinate them sequentially), the relatively high-dimensional input space, and the complexity of the environment itself. We believe this shows that using hierarchical architectures rather than monolithic networks and video game

environments rather than traditional tabletop robotics are plausible design choices for scaling up evolutionary robotics.

From the games perspective, creating a better performing bot (in the sense of killing better and scoring higher) is not a very interesting target as the currently available controllers are already able to outperform human players. Our controller is not by any means the best bot available for UT2004, but it was never meant to be, and considering the purposely limited input representation the result is still satisfying. A bot with access to the full game state can easily outperform our results, but our experiments show that FPS games can be a good testbed for system where the full representation of the world is not available, such as robotics.

We also believe that the architecture used can achieve more interesting and human-like behaviour compared to hardcoded bots using third-person environment representations; learning can be used to model human playing styles, the agent-centered inputs can let it react to the environment more believably. For example, the approach proposed in [23] could be used to imitate human playing styles, and the one proposed in [24] to create populations of interestingly different strategies.

While we have done many more experiments than would have been possible had we used physical robots, the complexity of UT2004 means that we have nevertheless been constrained by available computer power. Given more time, there's a number of obvious extensions to the current work:

- Execute runs of both the hierarchical and the monolithic controller evolution to prove the superiority of the former with statistical significance.
- Unfreeze the sub-controllers and continue the evolution of all parts of the controller simultaneously after obtaining a good behaviour selector, as was done in [13].
- Test the generalisation capabilities of our controllers by evaluating them on more maps and/or use several opponent bots.

In recent work, which will be soon be submitted for publication, we have slightly changed the representation of the sensors, tuned the learning process and redesigned the behaviour selector. Thus, we have been able to evolve controllers that significantly outperform the entry level UT2004 bot in death match score.

V. CONCLUSIONS

We described a hierarchical architecture for a bot in the modern FPS game Unreal Tournament 2004, and how the individual sub-controllers of this architecture were trained incrementally. In our experiments we showed that the evolved hierarchical controller solved the task better than a monolithic approach used as comparison. Furthermore, the hierarchical controller played the game quite well even though it was restricted to the type of agent-centered sensors that could theoretically be mounted on a robot. The proposed method could also be useful for automatically generating believable and interestingly different NPCs.

REFERENCES

- [1] D. Isla, "Probabilistic target tracking and search using occupancy maps," in *AI Game Programming Wisdom 3*. Charles River Media, 2006.
- [2] C. Overholtzer and S. Levy, "Adding smart opponents to a first-person shooter video game through evolutionary design," *aaai.org*, 2005. [Online]. Available: <http://www.aaai.org/Papers/AIIDE/2005/AIIDE05-028.pdf>
- [3] R. Kadlec, "Evolution of intelligent agent behaviour in computer games," *Master's thesis, Charles University in Prague*, p. 75, Sep 2008.
- [4] I. Karpov, T. D'Silva, C. Varrichio, K. Stanley, and R. Miikkulainen, "Integration and evaluation of exploration-based learning in games," *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2006.
- [5] M. Parker and B. D. Bryant, "Neuro-visual control in the quake ii game engine," *Neural Networks*, Jan 2008. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4634348
- [6] S. Priesterjahn, Kramer, A. Weimer, and A. Goebels, "Evolution of human-competitive agents in modern computer games," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2007.
- [7] C. Thureau, C. Bauckhage, and G. Sagerer, "Learning human-like movement behavior for computer games," *Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB'04)*, 2004.
- [8] N. Cole, S. J. Louis, and C. Miles, "Using a genetic algorithm to tune first-person shooter bots," in *Proceedings of the IEEE Congress on Evolutionary Computation*, 2004, pp. 139–145.
- [9] R. Small and C. B. Congdon, "Agent smith: Towards an evolutionary rule-based agent for real-time strategy games," pp. 1–7, Nov 2008.
- [10] J. Westra, "Evolutionary neural networks applied in first person shooters," *Master's thesis, Utrecht University*, Jan 2007. [Online]. Available: <http://people.cs.uu.nl/westra/articles/scriptie.pdf>
- [11] M. McPartland and M. Gallagher, "Creating a multi-purpose first person shooter bot with reinforcement learning," *IEEE Symposium on Computational Intelligence and Games*, 2008. [Online]. Available: <http://www.csse.uwa.edu.au/cig08/Proceedings/papers/8017.pdf>
- [12] R. Arkin, *Behavior-based robotics*. The MIT Press, 1998.
- [13] J. Togelius, "Evolution of a subsumption architecture neurocontroller," *Journal of Intelligent and Fuzzy Systems*, vol. 15, pp. 15–20, 2004.
- [14] F. Gomez and R. Miikkulainen, "Incremental evolution of complex general behavior," *Adaptive Behavior*, vol. 5, pp. 317–342, 1997.
- [15] T. Thompson and J. Levine, "Scaling-up behaviours in evotanks: Applying subsumption principles to artificial neural networks," in *Proceedings of the IEEE Symposium Computational Intelligence and Games (CIG)*, 2008.
- [16] S. Nolfi and D. Floreano, *Evolutionary robotics*. Cambridge, MA: MIT Press, 2000.
- [17] J. Denzinger, K. Loose, D. Gates, and J. Buchanan, "Dealing with parameterized actions in behavior testing of commercial computer games," *Proceedings of the IEEE 2005 Symposium on Computational Intelligence and Games CIG05*, pp. 37–43, 2005.
- [18] R. Adobbati, A. Marshall, A. Scholer, and S. Tejada, "Gamebots: A 3d virtual world test-bed for multi-agent research," *Proceedings of the Second International Workshop on . . .*, Jan 2001. [Online]. Available: <http://ironman.srv.cs.cmu.edu/~galk/Publications/01/gamebots.pdf>
- [19] R. Kadlec, J. Gemrot, O. Burkert, M. Bida, J. Havlicek, and C. Brom, "Pogamut 2 - a platform for fast development of virtual agents' behavior," *CGames07*, pp. 1–5, Oct 2007.
- [20] J. Elman, "Finding structure in time," *Cognitive Science*, vol. 14, pp. 179–211, 1990.
- [21] F. A. Gers and J. Schmidhuber, "Lstm recurrent networks learn simple context free and context sensitive languages," *IEEE Transactions on Neural Networks*, vol. 12, pp. 1333–1340, 2001.
- [22] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, pp. 182–197, 2002.
- [23] N. van Hoorn, J. Togelius, D. Wierstra, and J. Schmidhuber, "Robust player imitation using multiobjective evolution," in *Proceedings of the IEEE Congress on Evolutionary Computation (in press)*, 2009.
- [24] A. Agapitos, J. Togelius, S. M. Lucas, J. Schmidhuber, and A. Konstantinides, "Generating diverse opponents with multiobjective evolution," in *Proceedings of the IEEE Symposium on Computational Intelligence and Games*, 2008.