Combining Self Organizing Maps and Multilayer Perceptrons to Learn Bot-Behavior for a Commercial Game

C. Thurau, C. Bauckhage, and G. Sagerer
Applied Computer Science
Bielefeld University
P.O. Box 100131, 33501 Bielefeld, Germany
{cthurau,cbauckha,sagerer}@techfak.uni-bielefeld.de

ABSTRACT

Traditionally, the programming of bot behaviors for commercial computer games applies rule-based approaches. But even complex or fuzzyfied automata cannot really challenge experienced players. This contribution examines whether bot programming can be treated as a pattern recognition problem and whether behaviors can be learned from recorded games. First, we sketch a technical computing interface to a commercial game that allows rapid prototyping of classifiers for bot programming. Then we discuss the use of self organizing maps to represent manifolds of high dimensional data and how multilayer perceptrons can model local characteristics of such manifolds. Finally, some experiments in elementary behavior learning are presented.

INTRODUCTION

Throughout the last 30 years, computer games have undergone an astounding evolution. Compared to their ancestors in the 1970s, present day games create complex and atmospheric virtual worlds and thus convey a deep and haunting experience for the player. However, in contrast to graphics and physics simulation, programming intelligent behavior for artificial opponents (also called bots) did not proceed that fast. Rather, techniques applied here still mainly revert to ideas developed two decades ago (Cass 2002).

Up to now, actions and behaviors of bots usually are scripted or rely on (fuzzyfied) finite state machines (of admittedly complex structure). Plainly spoken, implementing bot behavior thus boils down to collecting huge sets of ‘if then’ rules. For many genres, however, this certainly does not reproduce the way experienced human players act. Consider for instance the (in)famous genre of first person shooter (FPS) games on which we will concentrate in this contribution. Already after a short period of practice players usually have learned how items are distributed on a map, how to cycle the map efficiently, and how to react to different actions of their opponents. I.e. while playing FPS games, humans tend not to think or plan. What seems like the outcome of anticipation or planning can be effectively reduced to reactive behaviors and strategies they gathered from experience.

In this paper, we propose to make use of this experience. The basic idea is to record many matches of human players (in gamers terminology such recordings are called demos) and to apply techniques from statistics, data mining, and pattern recognition in order to develop bots that imitate the behavior of individual players. To investigate this idea, we considered ID Software’s game QUAKE II® (which was chosen for experimentation because its source code is freely available and there are numerous demo resources). Assuming the state of a player at time $t$ to be given by a feature vector $\vec{s}_t$, his state at the next time step $t+1$ can be modeled to result from a function

$$\vec{s}_{t+1} = F(\vec{s}_t, \ldots, \vec{s}_{t-k}, \vec{\epsilon}_t, \vec{a}_t)$$

where $\vec{\epsilon}_t$ denotes environmental influences at time $t$ and $\vec{a}_t$ represents the player’s (re)action according to the history of his last $k$ states $\vec{s}_t, \vec{s}_{t-1}, \ldots, \vec{s}_{t-k}$. Hence, reactions can be understood to result from

$$\vec{a}_t = f(\vec{s}_{t+1}, \vec{s}_t, \ldots, \vec{s}_{t-k}, \vec{\epsilon}_t)$$

where the function $f$ might be learnable from training data. Although bot programming thus seems to be treatable as a problem of subsymbolic machine learning, surprisingly little efforts have been made in this direction. The growing body of literature on game bots still mostly deals with AI reasoning (Adobbati et al. 2001, Laird and Duchi 2000, Laird and v. Lent 2000) and known approaches to behavior learning from training data either consider computer vision applications or rather simplified games (Galata et al. 2001, Jebara and Pentland 1999, Pyeatt and Howe 1998, Sronck et al. 2002). But to the best of our knowledge, subsymbolic behavior learning for commercial games was not reported yet. A potential reason for this became apparent when we attempted to apply monolithic multilayer perceptors (MLPs) for behavior learning in QUAKE II®: dimensions and distributions of feature vectors turned out to be too high and too discontinuous for simple classifiers (Bauckhage et al. 2003).

Before we discuss a possible solution to this problem, the next section shall describe how state vectors can be extracted from QUAKE II® demos and how MATLAB® may be used as a rapid prototyping tool for bot behavior learning exper-
ments. Then, section 3 describes the use of self organizing maps (SOMs) to identify the intrinsic dimension of demo data and section 4 presents results in simple behavior learning obtained from hybrid coupling of SOMs and MLPs. Finally, a conclusion and an outlook will close this contribution.

**LINKING MATLAB® TO QUAKE II®**

QUAKE II® demo files are records of the network traffic between a player and the server he was connected to. Since QUAKE II® is able to fully reproduce whole game scenes from demo files, it is safe to say, that a single demo file contains every state \(s_t\) and reaction vector \(\delta_t\) of the observed player in a given match.

To give a little insight in demo files, it is necessary to have a look at how QUAKE II® network transmission is handled. A player's origin \(\bar{o}_{x,y,z}\) and viewangle \(\bar{v}_{yaw,pitch,roll}\) are just a few but important variables in a QUAKE II® network packet. Since a network packet sent from a server contains all relevant information regarding the current game state, it can be interpreted as a "world" state vector. However, when it comes to static or moving objects (flying rockets, health packages...) QUAKE II® demos do not include any visual information about object shape and are merely referencing entities, which are labeled and fully described inside local gamedata files.

A sideffect of this kind of information transmission is, that object detection is not necessary at all, as global knowledge about the type and state of every object is provided. Almost the same applies to a players reaction vector \(\delta_t\) at a given time \(t\). It can be directly extracted from recorded matches, since it is represented as a client-server packet block.

Although the state vector gained in a first parsing step is complete and depicts a whole game scene, it might be useful to further enhance it. Bump sensors for collision detection or range sensors for a more local position estimation are useful to further enhance it. Bump sensors for collision detection or range sensors for a more local position estimation are useful to further enhance it. Bump sensors for collision detection or range sensors for a more local position estimation are useful to further enhance it.

SELF ORGANIZING MAPS

Accurately analyzing a collection of high dimensional data suffers from the curse of dimensionality: the required number of samples grows exponentially with the dimension of the data points. However, for most practical problems \(n\) dimensional samples do not fill a \(n\) dimensional volume but reside on a \(m\) dimensional manifold where \(m \ll n\). Self organizing maps (SOMs) provide a means to identify and represent such manifolds (Kohonen 2001, Ritter et al. 1992). Next, we thus will briefly summarize some essentials of SOMs.

Given a \(n\) dimensional vector space \(V^n\), a SOM consists of a collection of vectors (called *codebook*) \(\bar{w}_r \in V^n\) where \(r\) denotes the coordinates of \(\bar{w}\) on a given topological structure (usually a \(m\) dimensional lattice with \(m \ll n\)). In a training phase from time \(t = 1\) until \(t = t_e\), feature vectors \(\bar{x} \in V^n\) are presented to the SOM and \(\bar{w}_{s}(x)\) is determined where

\[
s(\bar{x}) = \arg\min_r ||\bar{x} - \bar{w}_r||
\]

i.e. \(\bar{w}_s\) is the codebook vector closest to \(\bar{x}\). Then \(\bar{w}_s(t)\) is updated according to

\[
\bar{w}_s(t + 1) = \bar{w}_s(t) + \Delta \bar{w}(t)
\]

where

\[
\Delta \bar{w}(t) = \eta \cdot (\bar{x} - \bar{w}_s(t))
\]

i.e. \(\bar{w}_s\) will slightly move towards \(\bar{x}\). A similar update happens to all \(\bar{w}_r\) that are topologically adjacent to \(\bar{w}_s\). I.e. for all lattice coordinates \(r\) close to \(s\) we have

\[
\Delta \bar{w}_r(t) = \eta \cdot h_{rs} \cdot (\bar{x} - \bar{w}_s(t))
\]

where \(h_{rs}\) is usually given by

\[
h_{rs} = \exp\left(-\frac{|r - s|^2}{2\sigma^2(t)}\right)
\]

i.e. the displacement of \(\bar{w}_r\) depends on its lattice-distance to \(\bar{w}_s\). Since this algorithm causes adjacent points \(r, r'\) on the lattice to be assigned to adjacent points \(\bar{w}_r, \bar{w}_{r'}\) in the feature space, it realizes a topology preserving mapping from a high dimensional space to a lower dimensional manifold.

As an example, Fig. 1 shows a 3D projection of ten 8D codebook vectors arranged on a 1D lattice unfolding into the manifold that contains the presented training vectors.

In our experiments we clustered our training samples in state space using self organizing maps. In a second step of training, each cluster was assigned two multi-layer perceptrons that were trained using the data in the corresponding cluster, one for viewpoint, the other for velocity adjustment. To model time series context, we might have used time-delayed-neural-networks (Bauchhage et al. 2003) or recurrent networks. Since our training set did not include context critical player reactions, we reverted to classical MLPs.

The MLPs finally map our state vectors \(\bar{s}_t\) to an appropriate player reaction \(\delta_t(s_t)\). Whenever a bot’s reaction to a given situation \(\bar{s}_t\) has to be generated, the MLP with the associated highest SOM Neuron activity will be selected.
We tried to investigate the importance of our self organizing map topology and how it influences training results and overall classifier performance. Classifier performance was measured by computation of the mean squared error \( E_d \) of given data set \( d \), containing \( n \) state \( s \) and reaction \( \vec{a}^{\text{desired}} \) vector pairs.

\[
E_d = \frac{1}{n} \sum_{i=1}^{n} \| \vec{a}_i^{\text{desired}} - \vec{a}_i(s_i) \|^2
\]

To determine, how the number of implicitly encoded behaviors coheres with the overall performance of our classification system, we were using two different sets of training/test samples. In the first set, a movement path had to be learned. In the second set, aiming was introduced as an additional behavior to the movement behavior. Both behaviors were implicitly encoded in demo files, showing a human player aiming at an opponent and running around the map.

The player’s state vectors \( s_i \) at a given time \( t \) were used for training a SOM and served as an input vector for each associated classifier. We limited the world state vector to the player position \( \vec{x} \in \mathbb{R}^3 \), his distance \( d \in \mathbb{R} \) to the nearest opponent, and the vertical \( \theta \) and horizontal \( \varphi \) angles to this opponent. Although this leaves the bot with a cheap view of what else is going on, it is sufficient for an in-depth look onto the behaviors introduced in our two training sets.

The player’s reaction spectrum was also reduced to what is required for our particular task, adjustment of the player’s viewangle and movement speed. Viewangle adjustments are coupled in a 4 dim. vector, containing the player’s \( \text{YAW} \in [0^\circ, 180^\circ] \) and \( \text{PITCH} \in [0^\circ, 90^\circ] \) angles, each being assigned a signum \( \sigma(\text{YAW}), \sigma(\text{PITCH}) \in \{-1, 1\} \), in order to cover the full range. The player’s velocity is represented by a 2 dimensional vector, containing \( v_x \in [-400, 400] \) and \( v_y \in [-200, 200] \). Fig. 2 illustrates the bot’s perception and reaction spectrum. Therefor, we realized our agent by means of at least 2 classifiers, one responsible for viewangle adjustment with a 4 dimensional output vector, the other for player velocity control with a 2 dimensional output vector. Although we treated viewangle and velocity adjustment as equal important player reactions, in-game evaluation showed that viewangle adjustment has a greater influence on overall bot performance; it is also harder to learn. A mediocre performance for velocity classifiers does not necessarily lead to poor bot behavior. Consequently, viewangle adjustment has to be seen as the main player reaction to be improved.

All MLPs were trained using the Levenberg-Marquardt backpropagation algorithm and contained 6 neurons in their hidden layer. For improved accuracy, we eliminated MLP inputs where minimal and maximal values in a given training subset were equal, thus containing no relevant information.
for that particular MLP. As a consequence, the input vectors size of our MLP varied among clusters.

**Experiment 1:** Our first set of training samples showed a player running various routes around a map, and thus implicitly encoded a running behavior. The demos we used for training contained a total of 6970 state/reaction vector pairs, our evaluation set contained 1028 test samples. In this experiment, partitioning our training samples reduced the learning task for every MLP by lowering their responsibility to just a subset of possible state vectors, decreasing their number of \((\vec{s}_t, \vec{a}_t)\) pairs to be learned. Although the SOM is switching between the different MLPs, it is not really switching different behaviors. Every MLP represents the same behavior, only that it is specialized on a certain part of the state space. Our experiments showed some interesting results here, which can be seen in Fig. 3 and Tab. 1. Increasing the number of SOM neurons did lower our training mean squared error, but not as dramatically as we expected it. The same applies to our evaluation results, which tended to improve with an increasing size of neurons and associated MLPs. However, a single MLP still showed very good performance and was quite capable of learning the behavior encoded in our training set. A distribution of training samples, belonging to the same behavior, does not necessarily improve overall classifier performance.

**Experiment 2:** The second set of training samples included an aiming behavior, besides a single map run. An overall number of 2362 training samples had to be learned. For evaluation we used a set of 1599 test samples. In case of different behaviors encoded in our training set, a SOM automatically switches the appropriate behavior to a given state \(\vec{s}_t\), a capability which emerges from input space clustering. Here, increasing the number of SOM neurons had a huge influence on our training and evaluation results, the overall results can be seen in Fig.4 and Tab.2. In case of a SOM with just one neuron, thus training a single MLP for viewangle and velocity adjustment, it showed, that the MLPs were not able to learn the desired behaviors (or at least not satisfyingly). It is interesting to note that the MLPs failure did not depend on training set size, which was considerably smaller than in our first series of experiments, nor on the incapability of learning an aiming behavior (which can be done). The failure seemed rather to depend on the difficulty of learning different behaviors in a single classifier. With two neurons in a SOM, our state space partitioned in two subsets, leading to specialized MLPs for both behaviors and greatly improving the \(E_{\text{train}}\) and more important \(E_{\text{test}}\). A further increasing SOM size did improve the classifier performance in some cases, but again, even one classifier per behavior showed good results, which could not be improved very much by means of increasing the number of SOM neurons.

When it comes to in-game evaluation, the excepted bot performance could be watched in a realtime game. The classifiers, which already showed good offline evaluation results, made up for a much better in-game performance, showing some nice moves and a good aiming, once an opponent entered their view. Aiming did not seem to be perfect though, but the training data wasn’t perfect either.

### Table 1: Summary of training and offline evaluation results in viewangle and velocity adjustment, when learning a movement behavior.

<table>
<thead>
<tr>
<th># SOM neurons</th>
<th>(E_{\text{train}})</th>
<th>(E_{\text{test}})</th>
<th>(E_{\text{train}})</th>
<th>(E_{\text{test}})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Viewangle</td>
<td>Viewangle</td>
<td>Velocity</td>
<td>Velocity</td>
</tr>
<tr>
<td>1</td>
<td>0.340</td>
<td>0.224</td>
<td>0.141</td>
<td>0.058</td>
</tr>
<tr>
<td>2</td>
<td>0.173</td>
<td>0.136</td>
<td>0.040</td>
<td>0.025</td>
</tr>
<tr>
<td>4</td>
<td>0.105</td>
<td>0.125</td>
<td>0.077</td>
<td>0.015</td>
</tr>
<tr>
<td>6</td>
<td>0.149</td>
<td>0.137</td>
<td>0.092</td>
<td>0.010</td>
</tr>
<tr>
<td>10</td>
<td>0.088</td>
<td>0.121</td>
<td>0.076</td>
<td>0.081</td>
</tr>
<tr>
<td>20</td>
<td>0.203</td>
<td>0.117</td>
<td>0.063</td>
<td>0.064</td>
</tr>
<tr>
<td>30</td>
<td>0.106</td>
<td>0.133</td>
<td>0.062</td>
<td>0.017</td>
</tr>
</tbody>
</table>

### Table 2: Summary of training and offline evaluation results in viewangle and velocity adjustment, when learning a movement and aiming behavior.

<table>
<thead>
<tr>
<th># SOM neurons</th>
<th>(E_{\text{train}})</th>
<th>(E_{\text{test}})</th>
<th>(E_{\text{train}})</th>
<th>(E_{\text{test}})</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Viewangle</td>
<td>Viewangle</td>
<td>Velocity</td>
<td>Velocity</td>
</tr>
<tr>
<td>1</td>
<td>2.294</td>
<td>2.442</td>
<td>0.137</td>
<td>0.053</td>
</tr>
<tr>
<td>2</td>
<td>0.144</td>
<td>0.260</td>
<td>0.061</td>
<td>0.086</td>
</tr>
<tr>
<td>4</td>
<td>0.157</td>
<td>0.218</td>
<td>0.049</td>
<td>0.051</td>
</tr>
<tr>
<td>6</td>
<td>0.631</td>
<td>0.623</td>
<td>0.225</td>
<td>0.449</td>
</tr>
<tr>
<td>10</td>
<td>0.185</td>
<td>0.315</td>
<td>0.134</td>
<td>0.221</td>
</tr>
<tr>
<td>20</td>
<td>0.203</td>
<td>0.388</td>
<td>0.112</td>
<td>0.070</td>
</tr>
<tr>
<td>30</td>
<td>0.177</td>
<td>0.289</td>
<td>0.118</td>
<td>0.088</td>
</tr>
</tbody>
</table>

**CONCLUSION AND FUTURE WORK**

This paper reported about a MATLAB® interface to QUAKE II® that facilitates the examination of different pattern recognition techniques for bot programming. Given training sets of recorded games, functions that map the current state of the player’s character onto a reaction can be learned. As the corresponding data spaces are rather high dimensional and just sparsely covered by the training data, we discussed the idea of using self organizing maps to represent the manifolds on which player states are distributed. By means of multilayer perceptrons attached to the neurons of a SOM, local mappings from a state vector to a reaction vector can be realized. And indeed, several experiments with different hybrid neural network architectures indicate that it is possible to realize bots which behave human-like simply because they learned from human-generated training data.

Currently, we extend our approach to more complex behaviors. This includes efforts in online learning as well as the investigation of more sophisticated neural network architectures like hierarchical SOMs or mixtures of experts. Second, we plan to explore the appropriateness of other classifier techniques like support vector machines, decision tree methods or particle filtering. Also, with an increasing behavior complexity adequate datamining methods for feature selection need to be considered.
Figure 3: 3(a) Training performance when learning to run with a varying self organizing map size. 3(b) Offline evaluation results.

Figure 4: 4(a) Training performance when learning to run and aim with a varying self organizing map size. 4(b) Offline evaluation results.

ACKNOWLEDGEMENTS

This work was supported by the German Research Foundation (DFG) within the graduate program “Strategies & Optimisation of Behavior”.

REFERENCES


