Using Hierarchical Machine Learning to Improve Player Satisfaction in a Soccer Videogame

Brian Collins and Michael Rovatsos

School of Informatics
The University of Edinburgh
Edinburgh EH8 9LE
United Kingdom
BrianC@BrianLCollins.com
mrovatso@inf.ed.ac.uk

Abstract. This paper describes an approach to using a hierarchical machine learning model in a two player 3D physics-based soccer video game to improve human player satisfaction. Learning is accomplished at two layers to form a complete game-playing agent such that higher-level *strategy* learning is dependent on lower-level learning of basic *behaviors*. Supervised learning is used to train neural networks on human data to model the basic behaviors. The reinforcement learning algorithms Sarsa(λ) and Q(λ) are used to learn overall strategies mapping game situations to these basic behaviors. We compare learning and non-learning agents and provide game results. Performance in self-play is analyzed to obtain a deeper understanding of the agent's learning performance. Seventy people participated in a survey in which the learning agent led to a more dynamic and entertaining experience, while the non-learning agent was a slightly more difficult opponent.

1 Introduction

A problem often experienced in video games is that the results obtained by directly specifying artificial player ("agent") behavior may be very predictable or just not very good. Machine learning (ML) [1] offers many techniques that can be applied to games to create more dynamic and realistic AI agents that can adapt to new situations. In an attempt to increase entertainment in games, we apply a layered ML architecture to a 3D physics-based soccer video game. This involves learning lower level behaviors to facilitate the learning of successively higher level behaviors.

An additional benefit of learning is that it makes the design of artificial game players methodologically easier. Without ML, a programmer needs to manually "search" for and compare strategies used by agents. Considering the current availability of low-cost computational power, machines are much more suited to this task and can find better strategies in less time. Unfortunately, learning a model that directly maps inputs to outputs is too complicated for most games. Even if it were possible, it would probably not lead to a good general playing strategy, since it would be very difficult to manually adjust such a model and correct poor behavior. Layered ML alleviates these problems and allows for more control over the learning process. A custom hierarchy of behaviors to be learned can be designed specifically for a game, allowing the benefits of problem domain knowledge to be fully realized. Behaviors at any level of the hierarchy can either be hand-designed or learned. Using a layered learning model is flexible and allows for the learning of complex behaviors.

The main goal of the research presented here is to implement learning agents with acceptable in-game performance against people and other agents. Furthermore, agent behavior should be "human-like", i.e. appear natural and dynamic. Ultimately, we wanted to provide an entertaining game experience. To do this, we used neural networks to model low level behaviors based on human data. We used reinforcement learning (RL) to learn high level strategies based on these behaviors.

Stone applied a similar layered learning architecture to simulated robotic soccer [2], but the behaviors to be learned were different; entertainment and "human-like" behaviors were not required. An alternative method to learn based on human data in games is presented in [3], where rules are learned rather than neural networks. Another way to learn and use rules in games is presented in [4].

This project is a case study of considerable size examining the potential for using layered learning in video games to increase their overall entertainment value. We identified human behaviors in simple training games that were easier to model than others. We compared the performance of different RL algorithms [5] in a fairly complex environment, which exceeds the complexity of toy environments often used in the literature. Given enough time for learning, agents using learning consistently won games against agents using the best non-learning agents. Survey participants found the best non-learning agent to be more difficult, while the best learning agent was more dynamic. Overall, people found playing against the learning agent to be a more entertaining experience.

2 The Soccer Game

The problem domain is a 3D real-time one-on-one soccer game (screenshot in Fig. 1). The physics engine supports spheres with linear and angular momentum, as well as stationary planes, cylinders and boxes. Relevant physical equations are solved numerically in real-time. Players are represented as spheres and the soccer ball is a smaller sphere. Each player scores a point for a goal and spheres can collide with the walls. To control a player, a human or an automated agent provides a two-dimensional vector representing acceleration. They have the option to kick the ball at any time, releasing it in a straight line. Physical objects in motion are gradually slowed by friction and drag. Players acquire the ball by colliding with it; then the ball is attached to the player and is automatically repositioned in front of the player as they move. The ball can be stolen from an opponent by touching the ball then quickly getting it away from them.

The game is complex enough that it is highly improbable that an agent exists which performs optimally in every situation. Hand-designed agents are not very

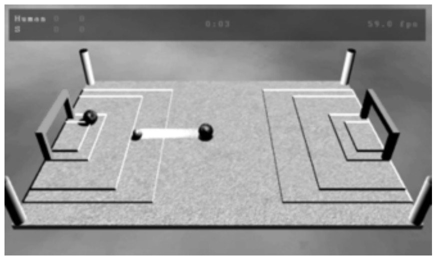

Fig. 1. The Soccer Game - A player is about to score a goal. The soccer field is flat and the white lines are zone markers used in representing strategies.

likely to be optimal. Optimal agents would need to be able to adapt their strategy to a wide range of situations. Low level behaviors, such as moving across the field can be difficult to specify or learn. The physics engine works with fairly complex differential equations (for friction, drag, angular momentum, ball handling, etc). Objects with angular momentum do not move in straight lines and non-zero velocities never remain constant. As a consequence, obtaining optimal low-level behaviors is a difficult process. For example, using acceleration to simultaneously control velocity and position is difficult within the game.

3 Layered Learning Architecture

We developed a two-layered learning architecture for this game. The lower layer allows for learning "basic behaviors" that are designed to accomplish simple tasks. The behaviors were chosen to be simple and robust, yet complex enough to be non-trivial. Each behavior needs to represent an opportunity for ML. An individual player's strategy can be formed by selecting a single basic behavior to use at a given time. Given the state of the world and a set of available behaviors, it is not trivial to select an appropriate one for a situation. Therefore, the higher "strategy learning" layer uses RL to learn which basic behavior to apply in each state from experience using information about rewards obtained previously. In other words, the higher layer learns a meta-strategy over the basic behaviors learned by the lower layer.

3.1 Supervised Learning of Low-Level Basic Behaviors

Seven basic behaviors were chosen for the game and we created a corresponding "training game" for each one. These training games are used to obtain data from human players and to objectively measure learning performance, the focus being on achieving "human-like" behavior at this level, rather than optimal performance in terms of the game in question. Supervised learning is used, since desired output vectors are known for the input vectors. During training data collection, a person plays the game and the game state is recorded at a fixed rate. The games are designed to present the player with random situations from the set of all game situations where the behavior could potentially be used. The aim is for a generalized representation of a basic behavior to be distilled from data obtained from the training games. Although this process cannot be guaranteed to work, very good results were obtained.

The basic behaviors chosen for the game are as follows: intercepting the ball (*Intercept*), retreating to the goal (*Retreat*), attempting to get close to the potential path of a kicked ball (*Defend*), aiming the ball toward the opponent's goal (*LineUp*), advancing toward the opponents goal (*Advance*), stealing the ball (*Steal*), and preventing an opponent from stealing (*KeepBallSafe*). Each of these are available to the higher learning layer as a primitive action with the exception of *LineUp*, for which three variations are provided to aim the ball toward different sections of the goal, thus allowing for a greater variety of strategies. Finally, there is a *Kick* behavior that does not require learning in this game.

The design of the training games has a fundamental effect on the behaviors that are learned. For example, it is possible that high training game performance may not be related to high performance against humans at the same task. For some behaviors, artificial opponents are needed in the training games to provide a consistent measurement of performance. In most training games the opponents select randomly from a set of simple hand-designed behaviors.

At the "basic behavior" layer, feedforward fully connected Multi-Layer Perceptrons (MLPs) were used to model and learn the behaviors. We used MLPs with a single sigmoid (tanh) hidden layer and linear output units, which can represent any bounded continuous function to an arbitrarily small error [6] and learn such functions from training data. The training algorithm used was stochastic gradient descent backpropagation with momentum [1] and training was off-line, so that the MLPs are fixed during actual games. A mean squared error (MSE) function was used to measure how well a given MLP models a training dataset. The neural networks for each basic behavior have two real number outputs (for 2D acceleration) and between six and ten real-valued inputs that are derived from game state information. For optimal performance, network inputs are first scaled to have a mean and variance of approximately 0 and 1, respectively. Each MLP had ten hidden neurons and approximately 100 real-valued weights to be learned. This appeared to be a good MLP size for the amount of available training data. The networks were trained for between 100 and 20,000 epochs. In some cases, MLPs with the best performance were found quickly and further training did not yield better results. In other cases, in-game performance continued to improve after thousands of epochs.

Twenty percent of the collected training data was used as a validation dataset. We also applied early stopping, where training is stopped in the event that the validation error has not improved for a certain number of epochs, i.e. allowing training to continue was not beneficial. The most consistent results were obtained when the validation data was taken from several disjoint segments of the collected data.

3.2 Reinforcement Learning for Higher-Level Strategies

A policy for playing the game can be defined to be a mapping from game states to actions (basic behavior implementations). The class of RL algorithms applied here is that of temporal difference (TD) methods [5], which learn from experience by continually improving an estimate of the optimal policy, i.e. the policy that leads to maximum long term reward. They are model free in the sense that they do not make use of explicit models of the environment (e.g., transition probabilities between states). Bootstrapping allows faster learning and updates estimates of reward based on other estimates. In terms of learning strategy we used on-line learning, where an agent learns while they play as this has the potential to lead to a dynamic experience for a person playing against the agent. However, this adaptiveness comes at the price of not being able to guarantee convergence of learning since a person's changing strategy leads to a dynamic environment.

However, for a fixed-strategy opponent, a policy learned using RL will, under certain conditions, converge to the optimal policy [5]. The conditions necessary for convergence were not created in the system, since it may not be desirable within a videogame; policies cease to be adaptive when convergence occurs. Despite this, it is important to note that minor changes to the RL implementation could allow for guaranteed convergence (such as decreasing the exploration rate over time such that it becomes zero in the limit).

RL heavily relies on an appropriate definition of state and action spaces, whereby states can be complex structures built up over time that implicitly contain information about past occurrences and the requirement is that all information needed to make a decision is implicitly in the knowledge of the current state. To learn a strategy in a reasonable amount of time, the most important information must be encoded within a small set of states. Additionally, in the case of on-line learning in a video game, learning has to happen within a reasonable amount of time as game play only extends over a limited period and human players need to be able to experience their artificial opponent's adaptiveness to make the game appear interesting.

The state space defined for our domain was based on quantized values of the most important information, such as the player's proximity to the soccer ball and some historical information about recent ball possession. Without any historical information, an agent may observe that they do not have the ball immediately before they score a goal and may learn that not having the ball is good. Rewards are given for goals and negative rewards are given when the opponent scores. Rewards are also given in some other situations to increase the speed of learning, for example when the player gains possession of the ball or loses it.

The RL algorithm used was the standard version of Sarsa(λ) [5]. This algorithm uses so-called eligibility traces to determine how each observation can be used to update predictions made in the past. The result is a table of estimated Q-values, which estimate the expected long term reward for taking an action in a state. A variant of Q(λ), another algorithm which estimates Q-values, referred to as "naive Q(λ)" in [5] was also implemented, but the time needed for learning was much longer. The one-step Sarsa method (equivalent to Sarsa(0)) performed significantly better than one-step Q-learning. The likely cause is that Q-learning needs more time to implicitly model state transitions. Sarsa takes actual state-action sequences into account and learning was faster.

A Q-table alone does not provide enough information for selecting an action to apply in a given state. There is always a trade-off between exploration and exploitation of current knowledge. Exploitation may lead to higher rewards, while exploration may lead to increased knowledge and ultimately higher rewards. Epsilon-greedy (ε-greedy) exploration and Softmax exploration [5] were both implemented and Softmax provided better performance. It includes an exploration temperature τ, which allows for a blend of exploration and exploitation. For high temperatures, actions are essentially selected at random. For low temperatures, actions with low Q-values will rarely be selected, while there is a high probability of selecting an action with a relatively high Q-value. This focuses exploration on actions that are estimated to be nearly optimal.

4 Experiments

4.1 Basic Behaviors

In all training games besides *Advance*, we were able to find a neural network which consistently performed better than a human expert. Note that performance is indicated by training game score. For some behaviors, the ideal outcome occurred and the networks with the lowest validation dataset error had the highest training game performance. An example of this is the *Intercept* behavior in Fig. 2. For other behaviors such as *Retreat* (Fig. 2), the highest performing networks were found earlier in the training process while validation set error was still dropping. Figure 2 shows how validation set error was slightly higher than training set error throughout the training process. The correlation of training game performance to data set error varied between behaviors, but was strong for the simplest behaviors.

In two cases, obtaining high performance networks through training was particularly difficult. It is possible that not enough high quality training data was obtained. Low dataset error cannot guarantee high in-game performance, but the process worked well for most of the behaviors. Using highly processed information from the game state as input to the networks was not effective, since relationships between inputs and outputs became more complex and more difficult to learn.

Hand designed basic behavior implementations were created with varying amounts of effort and manual tuning. In two cases, the MLP implementations performed better than the hand designed alternatives. For these two networks, the correlation of training game performance to data set error was strong and networks with lower dataset error performed better in the game. In other cases, MLP performance was comparable to hand-designed behavior performance. In all instances the MLP based behaviors appeared to be more “human-like” and less rigid than their hand-designed counterparts. Noise could be added to the outputs of the hand-designed behaviors, but results are not likely to be as good. Human actions in the game can be imprecise and the MLPs implicitly model the types of deviations a player might make from their desired paths.

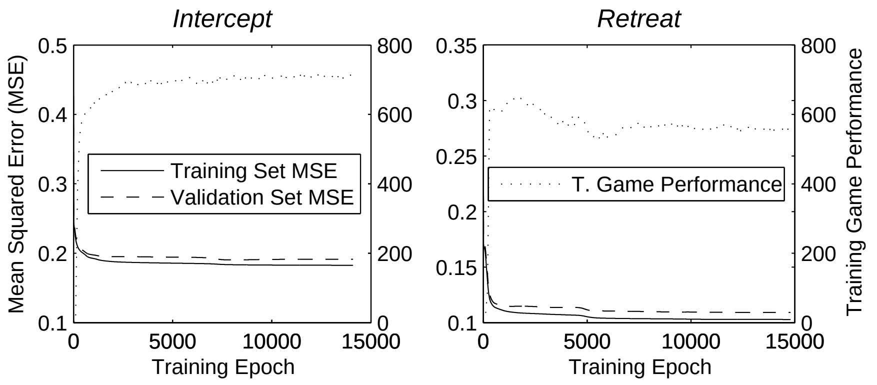

Fig. 2. The neural network training process for two basic behaviors. Actual training game performance is not necessarily related to mean squared error on the validation dataset.

4.2 Higher-Level Strategies

Experimentation was performed to find good values for the many parameters of the Sarsa(λ) algorithm. After reasonable initial values were selected, many games were run against an agent with a hand-designed deterministic strategy to optimize the parameters. A state space with 108 states and 432 state-action pairs was used, which led to high performance policies and required less learning time than larger state spaces. When the number of states used was 500, up to ten times more learning time was required to obtain the same results. After extended periods of learning, performance never exceeded the performance of the small state space. Increasing the number of states may be necessary when playing against opponents that have highly complex strategies.

For each experiment, two parameters were varied simultaneously in pursuit of optimal values. The λ parameter determines how much observed information is used to update past predictions and the best value was 0.7. Softmax exploration led to the best overall performance. Higher exploration temperatures, of around 2.0, led to increased performance at the conclusions of the sessions but relatively low performance throughout the sessions. Balanced exploration (temperatures ranging from 0.5 to 2.0) led to better overall results and decent performance at the conclusions of the sessions. The best value for the learning rate α was 0.03. The best values of the reward discount factor γ were between 0.925 and 0.98. The $\gamma\lambda$ product is closely related to the eligibility trace and intermediate values of this product, between 0.55 and 0.75, led to the best results.

4.3 Combining the Learning Layers

The best implementations of each type (ML/non-ML) for each learning layer were combined. Strategy implementations were: RL (Sarsa(λ) with Softmax exploration), a hand-designed stochastic policy, and a hand-designed deterministic policy (these policies are essentially based on finite-state machines). MLP based behaviors and hand-designed (HD) behaviors were tested in combination with each (high-level) strategy. Stochastic fixed policy agents won almost all of their games against deterministic ones.

Overall, the players using MLP based behaviors outperformed the counterparts that used HD behaviors. On average, agents using HD behaviors won 42% of their games while agents using MLP behaviors won 57% of them. This does not coincide with the training game performance of the behaviors. Some behavior implementations display weaknesses during game-play which cannot be seen in the training games. Perhaps the MLP behaviors are more robust despite slightly inferior (on average) training game scores. For games lasting 25 game-hours (simulated in around 3 minutes), both RL players won the majority of their games. They clearly outperformed all fixed policy agents. When the Sarsa/MLP player competed directly with the Sarsa/HD player, the Sarsa/MLP player won most games. It seems to be easier to learn a strategy based on the more complex MLP behaviors. Since the MLP behaviors are more robust, a higher proportion of strategies based on them (as opposed to HD behaviors) lead to acceptable performance. This increases the chances of finding an MLP based strategy through learning that performs consistently well. When self-play was simulated, two learning agents with initially no knowledge competed directly. Actions in such games were initially random. After some time, simple strategies such as heavily offensive strategies emerged. After enough time, complex and well-balanced strategies could be observed. Provided the exploration temperature τ was significantly high, around 0.5 to 2.0 (depending on the lower level behavior implementation), this process consistently generated useful and balanced strategies.

Learning times were reduced to one game-hour and Q-tables learned in various 25 hour games were used as initial Q-tables. Figure 3 illustrates some of these results. The players benefited from any initial knowledge, particularly when the exploration temperature τ in previous game was high, 2.0 in this figure. A lower τ in the next game of 0.5 allowed the players to immediately exploit their knowledge and simultaneously adapt it to the new situation. These are the τ values used for Fig. 3. Both RL players benefited the most from their experience against the stochastic policy agent. There was a 50% increase in the number of one hour games won against various (learning and non-learning) automated opponents. Experience from self play was beneficial to both RL players, leading to a 35% increase in games won. Each RL player also benefited from any experience obtained by the other, which lead to a 28% increase in games won. This indicates that the process can lead to generally good strategies. Self play is an interesting case, because it leads to a dynamic non-Markovian environment. RL for non-Markovian processes is still an active area of research. Results cannot be guaranteed [5], but good results were obtained leading to increased confidence in the underlying methods. The amount of time needed for learning is important for a real-time game. Good strategies were learned from no initial knowledge in less than an hour of game time, which is a reasonable amount of time for a person to play such a game. Simulated games can be run much faster than real-time to obtain prior knowledge, which allowed for significantly better results in the early stages of learning.

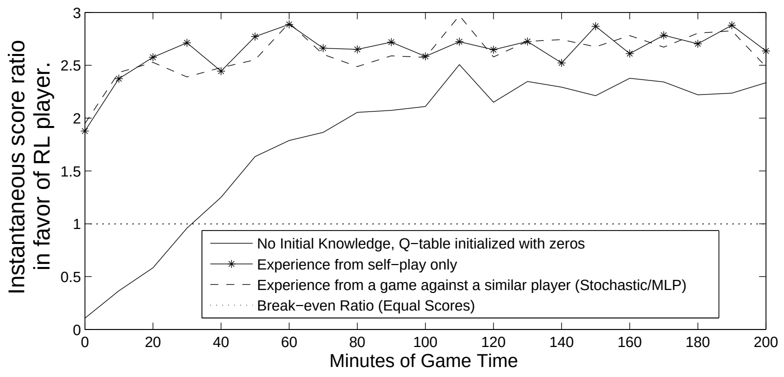

Fig. 3. Average performance of an RL agent as time increases. The agent had varying initial knowledge and used MLP based behaviors. The opponent was the stochastic/HD agent. The score ratio in favor of the RL agent is presented. This is an instantaneous ratio consisting of points scored within 10 game minutes. An agent with no initial knowledge began outperforming their opponent, who uses a fairly complex strategy, in around 30 game minutes on average. Knowledge from self-play is seen to be very useful.

4.4 Survey

In order to evaluate the more "human-centric" goals of the project, an online survey was conducted in which 70 people participated. Participants played five minute games against two different opponents in a random order. The ML player used Sarsa(λ) for the higher layer and MLPs for low-level behaviors. The non-ML player used a hand designed stochastic strategy and hand designed behaviors. People usually found the non-ML player to be slightly more difficult. In order to give the ML player a chance of providing a good challenge, it had been trained against the non-ML player and could defeat it consistently. People were not told which opponent used ML. An average of 41% of people won their first game, while 50% of people won their second game. This did not seem to depend on which opponent was encountered first. Overall, people found the non-ML player to have a better strategy and be more difficult. However, it was significantly easier for players to predict the actions of the non-ML player. In general, people felt that the ML player moved around more fluidly and behaved in a much more dynamic 'human-like' way. Some people commented that player behaved like a human who was not an expert at the game and still made some mistakes. Despite occasionally making mistakes, the ML player proved to be an entertaining and challenging opponent. People may have felt the non-ML player had a better strategy because its strategy is more consistent. The ML player sometimes appears to change its playing style several times during a game, which may have had a confusing effect on players not anticipating it. Overall, people were more satisfied with the ML player. Some noted that it adapted based on their own strategy. A representative comment about the ML player was that it “felt less ‘cookie cutter AI’ ... he sort of moved around more fluidly and didn’t resort to the same patterns.” Many people appreciated the adaptive nature of the ML player. Some others preferred the more traditional challenge offered by the non-ML player and the pleasure of finding a strategy that consistently outperformed this non-adaptive opponent.

5 Conclusion

In this paper, we described the implementation of a layered learning architecture in a soccer videogame. It was based (a) on learning basic behaviors based on data obtained from humans performing simple tasks and (b) on learning a game strategy that uses these basic behaviors as primitive actions by means of reinforcement learning.

At the lower level, our method was very successful for learning human-like, natural-looking behaviors, provided that the behaviors to be learned were simple enough. Attempts to learn more intricate behaviors in this way led to disappointing results, as training data was noisy and complex. The design of the training games used for data collection is very important at this stage. At the higher level, the RL implementation led to very good results with reasonable amounts of learning time. The ML agents performed well against other agents and were well received by human players, who found them dynamic and entertaining.

The hierarchical learning approach is very flexible and led to efficient learning. Much control over the learning process was retained and undesirable actions could be corrected with relative ease. The hierarchical learning model used could easily be extended to more interesting situations with multiple agents on each team. Various policies could be learned through RL for common roles in a soccer team, using different reward functions. In the future, we plan to add a third “social” learning layer which would enable artificial players to learn optimal teamwork coordination strategies.

References

1. Mitchell, T.M.: Machine Learning. McGraw Hill (1997)
2. Stone, P.: Learning in Multi-Agent Systems. PhD thesis, Computer Science Department, Carnegie Mellon University (1998)
3. van Lent, M., Laird, J.: Learning hierarchical performance knowledge by observation. In: Proc. 16th International Conf. on Machine Learning, Morgan Kaufmann, San Francisco, CA (1999) 229–238
4. Laird, J.E.: It knows what you’re going to do: adding anticipation to a quakebot. In Müller, J.P., Andre, E., Sen, S., Frasson, C., eds.: Proceedings of the Fifth International Conference on Autonomous Agents, Montreal, Canada, ACM Press (2001) 385–392
5. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)
6. Cybenko, G.: Approximation by superpositions of a sigmoidal function. Mathematics of Control, Signals, and Systems **2** (1989) 303–314