Player Modeling using Knowledge Transfer

Guy Shahine
DigiPen Institute of Technology
5001-150th Ave NE
Redmond, WA 98052
USA
E-mail: gshahine@digipen.edu

Bikramjit Banerjee
School of Computing
The University of Southern Mississippi
118 College Drive
Hattiesburg, MS 39406
E-mail: Bikramjit.Banerjee@usm.edu

KEYWORDS

Transfer learning, Player modeling, Influence diagrams

ABSTRACT

We propose a technique for creating reusable models of other agents (software or human) in a shared environment, with application to video game AI, and AI in general. In particular, we build upon an existing technique for agent modeling using *influence diagrams*, and propose a method to transform it into a reusable model. Such models can be used effectively in reasoning by AI characters (NPCs) for predicting the behavior of human players, for collaboration or competition in *different* tasks. We show experiments in two collaborative tasks (anti-air defense and predator-prey) that clearly demonstrate the reusability and efficiency of our modeling technique across different tasks.

INTRODUCTION

Knowledge transfer (Konidaris and Barto, 2007; Banerjee and Stone, 2007) is a relatively new technique in AI, that is useful in incremental learning of reusable skills and general knowledge. The key idea is to reuse knowledge acquired in previous tasks (called "source") to learn/solve new, but related tasks (called "target") in order to

- offset initial performance in the target task, compared to learning/solving from scratch
- achieve superior performance faster than learning from scratch, when considering learning problems

This new trend in AI research seeks to overcome the long-held tradition of developing specialized systems for given tasks. Instead, we seek to develop life-long learning systems that can map representations and knowledge efficiently from one task to another, and bootstrap performance in the newer tasks by exploiting previously acquired knowledge/skills from a different setting. A motivating example can be as follows: Suppose an agent (NPC) learns to coordinate with a human player in the task of jointly intercepting several incoming missiles; i.e., successful coordination will result in the minimal (perhaps 0) damage to assets from the missiles. Can this agent reuse any of this knowledge in coordinating with the same player in a *different* task, such as coordinated hunting of a prey? Traditional AI and Multi-agent systems literature treats the second task as a new task and embarks on learning a coordination policy from scratch.

The major attraction of knowledge transfer in game AI is the dramatic impact it can have on learning speed, which has so far proved to be a bottleneck for most mainstream machine learning technique in games. No matter what baseline learning technique we choose, knowledge transfer can make it practical in complex game scenarios by exploiting knowledge acquired from previous tasks.

In this paper we focus on the problem of learning models of other agents/players in a domain, and explore the beneficial impact of knowledge transfer on this problem. There may be many motivations for studying the problem of agent modeling. In collaborative domains where communication is expensive, unreliable, and/or prone to interception by adversaries, modeling can be an invaluable tool in predicting the behavior of team-mates, to act in a coordinated fashion (Carmel and Markovitch, 1996). Another major motivation for agent modeling comes from the field of games, especially video games itself. Prohibitive search cost has kept deep searches in the strategy space in game-trees (especially ones with large branching factors, such as chess, or any video game) outside the purview of all but the simplest of games. However, the ability to predict other agents' choices can greatly reduce the branching factor at the search-tree-nodes corresponding to the choices of the other agents. This could enable an agent to only focus on its choices and consequently, search deeper/faster.

We build upon a previous work by Suryadi & Gmytrasiewiez (1999) that outlines a simple technique for learning the decision function of another agent from observations, using an *influence diagram* model (see later section). We argue that the structure of this model does not lend itself to reuse. It is suited only for the problem at hand, and if we were to deal with the same agents/players in another task, we would need to acquire a new model in the new task, all over again. We then show how this model can be transformed into a reusable one, by decomposing the environment-dependent components into two finer parts – one of which is dependent on the agent (its intrinsic character assumed to be unchanged from one task to another; hence transferable across tasks) and the other still depends on the current task. We show that this decomposition allows the second part (task dependent) to be readily deducible from the environment, and may not need to be learned. As a result, our modeling approach enables reuse/transfer of models in many different tasks involving the same set of agents/players.

Essentially, we propose a technique for acquiring reusable models of other agents, through interactions in a variety of

tasks. While the tasks can be widely differing, we assume that the other agents/players (teammates or adversaries) are the same across different tasks. This assumption can be appropriate in many applications. For instance, in video games it is reasonable to assume that a Non-Playing Character (NPC) needs to collaborate or compete with the *same* human player(s) in different tasks, such as collaborative defusing of Improvised Explosive Devices (IEDs), joint patrolling, hunting etc. Consequently, we can

- personalize NPCs to human players, creating the illusion of another "player" rather than a software, and promoting empathy for NPCs,
- allow agents to incrementally model a human player through several tasks, creating an increasingly complete picture of the latter's preferences, styles etc.,
- allow reuse of agent's code and knowledge across different games, thus reducing development time. Human players often do not prefer novice opponents to sophisticated ones, every time he enters a new game. This is especially true when he himself is able to draw on experiences acquired from previous games.

RELATED WORK

Suryadi and Gmytrasiewiez (1999) train an agent to learn a model of another agent in a multi-agents system where the other agent can be either automated or human. The framework makes use of influence diagrams as a modeling representation tool and three strategies are used to create a new model of the other agent, which are based on learning its capabilities, beliefs and preferences, given an initial model and the agents' behavior data. Consequently, through the agents' behavior data, things might change where some capabilities are added or removed, and even some beliefs are altered but the task cannot be switched, i.e. we cannot move to another task while retaining what the agent has learned so far. Although we base our current work on this method, we provide significant extension for reusability of the learned model in other tasks.

Transfer learning has received most attention in the realm of reinforcement learning. The options framework defined by closed loop policies for taking an abstract action over a period of time provides an ideal foundation for knowledge transfer. Examples of options include picking up an object, going to lunch, and traveling to a distant city, that involve specific sequences of primitive actions that can be activated only in certain states. Options enable temporally abstract knowledge and action to be included in the reinforcement learning framework in a natural and general way (Sutton et al. 1999). Consequently, options framework provides methods for reinforcement learning agents to build new high level skills. However, since options are learned in the same state space as the problem that the agent is solving, they cannot be readily used in other problems/tasks that are similar but have different state space. Konidaris and Barto (2007) introduce the notion of learning options in *agent space* – the space generated by a feature set that is present and retains the same semantics across successive problem instances – rather than in *problem space*. Agent-space options can be reused in later tasks that share the same agent-space but have different problem-spaces. Our approach in this paper bears some similarities to this idea of agent-space since we essentially convert a task dependent model language into a player/agent dependent model language. In a recent work (Banerjee and Stone, 2007) we have provided a transfer learning method for reinforcement learning agents in the General Game Playing (GGP) domain. In this paper, we carry forward the lessons learned to impact a new domain, viz., player modeling for other genres of games (RTS, FPS etc.)

While humans effortlessly use experience from previous tasks to improve their performance at novel (but related) tasks, machines must be given precise instructions on how to make such connections. Roy and Kaelbling (2007) describe a Bayesian model based prediction scheme in a meeting-scheduling domain. They provide a hierarchical extension to a Naïve Bayes model that can incorporate data from many different users and make predictions for another user, thus exploiting previous experience. In contrast, we use an influence diagram model and instead of relying on model combination, we seek model decomposition to identify user/player specific components that can then be transferred to a new task. Furthermore, we study our technique in simulated game scenarios with an eye toward future application to video games.

INFLUENCE DIAGRAMS

An Influence diagram (Howard and Matheson, 1984) is a graphical scheme of knowledge representation for a decision problem. It may be viewed as an extension to a Bayesian (Charniak, 1991) or belief network (Pearl, 1988), with additional node types for decisions and utilities. Influence diagrams have three types of nodes: nature node (a.k.a. chance node), decision node, and utility node. Nature nodes correspond to decisions in nature, as in belief networks. They are represented in the diagram as ovals and associated with random variables or features, which represent the agent's possibly uncertain beliefs about the world. Decision nodes holds the choice of actions an agent has, thus represents the agent's capabilities and they are represented as rectangles in the diagram. The agents' utility functions are specified using utility variables, represented as diamonds in the diagram. The links between the nodes summarize their dependence relationships. Evaluation of the influence diagram is done by setting the value of the decision node to a particular choice of action, and treating the node just as a nature node with a known value that can further influence the values of other nodes. Conditional Probability Tables (CPTs) are associated with nature and decision nodes, giving $P(child \mid parents)$. An algorithm for evaluating influence diagrams can be found in Russell and Norvig (1995).

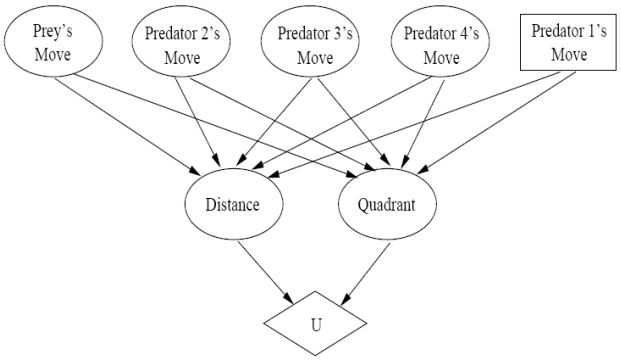

Figure 1: An Influence Diagram model for the Predator-prey task (described later)

Figure 1 above shows an influence diagram used as a basis in the predator/prey task discussed later. Here four predators are trying to jointly catch a prey by closing in on it and surrounding it. In this example, each of the predators 2 thru 4 decides on its next move by taking into consideration two factors: its distance from the prey and the current location of the other predators on the board (whether they are on the same quadrant or spread out, relative to the prey). Based on his observation of these factors, Predator 1 tries to make a prediction for each of the other predators using this influence diagram, and then selects an action that will lead to the best coordination with his predictions.

Below we present another example of a Multi-Agent Influence Diagram (MAID), although this is not a coordination task.

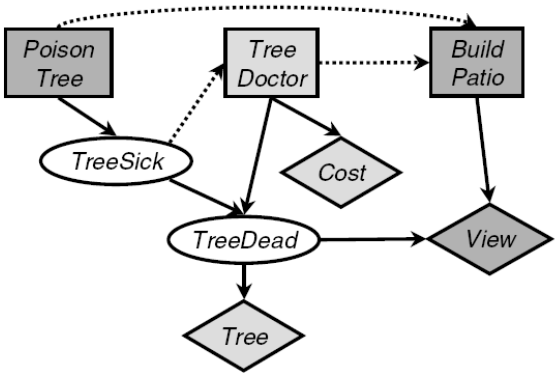

Figure 2: A MAID for the Tree Killer example; Alice's decision and utility variables are in dark gray and Bob's in light gray

In the Tree Killer Example (Koller et al., 2003), Alice is considering building a patio behind her house, and the patio would be more valuable to her if she could get a clear view of the ocean. Unfortunately, there is a tree in her neighbor Bob's yard that blocks her view. Being somewhat unscrupulous, Alice considers poisoning Bob's tree, which might cause it to become sick. Bob cannot tell whether Alice has poisoned his tree, but he can tell if the tree is getting sick, and he has the option of calling in a tree doctor (at some cost). The attention of a tree doctor reduces the chance that the tree will die during the coming winter. Meanwhile, Alice must make a decision about building her patio before the weather gets too cold. When she makes this decision, she knows whether a tree doctor has come, but she cannot observe the health of the tree directly. The Multi-Agent Influence Diagram (MAID) for this scenario is show in Figure 2.

CAPABILITIES

In the following, we use the words "learner", "modeler", and "agent" interchangeably, and refer to the target of modeling as "player", meaning the human player. As in Suryadi and Gmytrasiewicz (1999), a player's capabilities are formed by the set of actions ($a_i, i = 1,..., M$) that it can perform to interact with the world. These actions will be represented in the influence diagram as possible value assignments to the player's decision. Initially, only the known actions are available to form the CPTs, but if a player is observed to employ other (previously unknown) actions, the set can be expanded to include these new actions. Conversely, if the player somehow loses the ability to perform a certain action, that can be determined by its absence in the observation history, especially if the player consistently picks the next most likely action (according to the model that the agent maintains). The missing action can then be deleted from the player's capability.

In Figure 1, each of the predators can move to one of the four cells surrounding its current location. The modeler/learner (Predator 1) can formulate the CPTs for the influence diagram using empirical observations of its teammates' actual moves. Since some available actions may not be known (or pertinent) to the modeler, the latter should ideally account for such contingency using an "*Other*" action that bunches such actions together. For instance, if a certain predator can move to a diagonal neighboring cell unlike what the modeler expects, that capability may become crucial to coordination and needs to be accounted for. See Suryadi and Gmytrasiewicz (1999) for a more detailed discussion of handling agent capabilities in model building.

PREFERENCES

Each player has his own preferences depending on the problem, and takes into consideration a set of features which influence his decisions. Let $X = \{X_1, X_2, \ldots, X_N\}$ be such a set of features. In Figure 1, X is the set of parents of the utility node (U), i.e., the nodes *Distance* and *Quadrant*. As in Suryadi and Gmytrasiewicz (1999), we assume that utility is a linear function of these features

$$U = w_1 x_1 + \ldots + w_N x_N$$

where w_k are unknown weights measuring the influence of feature X_k on the utility, and x_k is a value of X_k. The assumption of linear contribution of x_k is a choice being made for simplicity; a different choice might be necessary for certain domains. E.g., if x_k stands for money, then

$\log(x_k)$ might be more suitable given the sub-linear nature of money's utility (St. Petersburg paradox).

It has been shown in Suryadi and Gmytrasiewiez (1999) that the optimal decision of the player based on the model is given by

$$A^* = \arg\max_{a_i} \sum_{k=1}^{N} w_k \chi_k^i \quad (1)$$

where a_i are the actions available to the player, and χ_k^i denotes the expected value of feature X_k given a_i and background evidence E, i.e.,

$$\chi_k^i = \sum_l x_{k,l} P(X_k = x_{k,l} \mid a_i, E)$$

where $x_{k,l}\,(l = 1,..)$ denote different possible values of X_k. The probabilities are available from the CPTs of the influence diagram and are assumed to have been already learned, or given. In the predator-prey example in Figure 1, the probability distribution is always pure (e.g., the distance between the next position of the predator after choosing movement direction a_i, and the current position of the prey has a single possible value), whereas it can be a mixed distribution in general.

Using equation (1), the modeler predicts that the player is going to execute action A^*, and verifies if this matches the actual action that the player chooses. If there is a mismatch then it should adjust the weights by applying a well-known gradient descent technique, viz., delta rule (Widrow and Hoff 1960). The idea is to minimize a cost function based on the error signal so that each weight adjustment brings the actual output value closer to the desired value. A commonly used cost function is the mean-square-error criterion:

$$E(t) = \frac{1}{2}\sum_{i=1}^{M} e_i^2(t)$$

where $e_i(t)$ is the error signal generated by comparing the agent's prediction and the player's actual decision. So

$$e_i(t) = d_i(t) - y_i(t)$$

where $d_i(t)$ is the player's decision at *t* and $y_i(t)$ is the agent's prediction.

Functionally, this model for weight learning can be considered to be a neural network where χ's are the inputs and $y_i(t)$ are the outputs, as shown in Figure 3. In this figure, v_i is the expected utility of the *i*th action, a_i, and the sigma (not the sigmoid function common in neural nets) module performs the maximization process of equation (1). Now according to the delta rule, the weight adjustment is proportional to the product of the error signal and the input unit. We require normalization of the χ before presenting them to the neural network, so that the resulting weights are normalized as well (Suryadi & Gmytrasiewiez, 1999). The normalization is given by:

$$\aleph_k^i = \frac{\chi_k^i}{\sqrt{\sum_{r=1}^{M}\sum_{s=1}^{N}\left(\chi_s^r\right)^2}}$$

Now,

$$\Delta w_k = -\eta \frac{\partial E(t)}{\partial w_k(t)} = -\eta \sum_{i=1}^{M} \frac{\partial E(t)}{\partial e_i(t)} \cdot \frac{\partial e_i(t)}{\partial w_k(t)}$$

$$\Delta w_k(t) = \eta \sum_{i=1}^{M} e_i(t) \aleph_k^i(t)$$

where $k = 1,...,N$ and η is a constant denoting the learning rate.

Neural Network Background

In this subsection, we provide a description of neural networks for the sake of clarity. Readers familiar with this machine learning technique may skip this subsection.

A neural network is made of basic units arranged in layers. The first layer is the input layer and in our case it is formed from several inputs (depending on the problem) represented by the normalized expected values $\aleph_k^i$. The last layer is the output, and in our model there is an output for each possible action, although only one (the modeler's prediction by equation (1)) will be activated at any time. The intermediate layers (if any) are called the hidden layers. The input information is fed to the first layer and then propagated to the neurons of the second layer for further processing. The result is propagated to the next layer and so on until the last layer is reached. The goal of the network is to learn or discover some association between input and output, or to analyze, or to find the structure of the input pattern. The learning process is achieved through the modification of the connection weights between units. These weights, called synaptic weights multiply (i.e. amplify or attenuate) the input information: A positive weight is considered excitatory, a negative weight inhibitory.

Each of these units is a simplified model of a neuron and transforms its input information into an output response. This transformation involves two steps: First, the activation of the neuron is computed as the weighted sum of it inputs, and secondly, this activation is transformed into a response by using a transfer function. Formally, if each input is denoted *xi*, and each weight *wi*, then the activation is equal to $a = \sum x_i w_i$ and the output denoted y is obtained as $y = f(a)$.

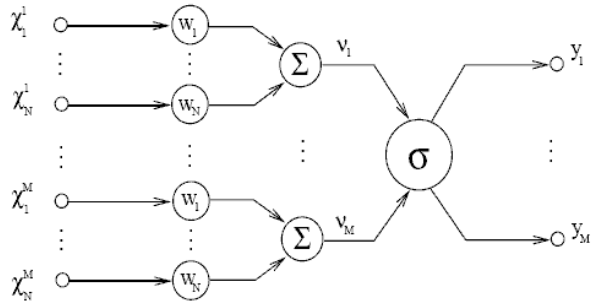

Figure 3: Neural Network Architecture

The architecture of the network, along with the transfer functions used by the neurons and the synaptic weights, completely specify the behavior of the network. Our specific architecture is shown in Figure 3. Notice that the neural units are expanded for clarity and that there is no hidden unit.

Neural networks are adaptive statistical devices. They can change iteratively the values of their parameters (i.e., the synaptic weights) as a function of their performance. These changes are made according to learning rules which can be characterized as supervised (when a desired output is known and used to compute an error signal) or unsupervised (when no such error signal is used).

The Widrow-Hoff rule (1960), a.k.a. gradient descent or Delta rule, is the most widely known supervised learning rule. It uses the difference between the actual output of the network and the desired output as an error signal for units in the output layer. Units in the hidden layers cannot compute directly their error signal but estimate it as a function (e.g., a weighted average) of the error of the units in the following layer. This adaptation of the Widrow-Hoff learning rule is known as error backpropagation. With Widrow-Hoff learning, the correction to the synaptic weights is proportional to the error signal multiplied by the value of the activation given by the derivative of the transfer function. Using the derivative has the effect of making finely tuned corrections when the activation is near its extreme values (minimum or maximum) and larger corrections when the activation is in its middle range. Each correction has the immediate effect of making the error signal smaller if a similar input is applied to the unit. In general, supervised learning rules implement optimization algorithms akin to gradient descent techniques because they search for a set of values for the free parameters (i.e., the synaptic weights) of the system such that some error function computed for the whole network is minimized (Abdi et al. 1999).

OUR CONTRIBUTION

Notice that the factors (viz., distance, quadrant in Figure 1) that affect the player's utility (and hence his decision) change from one problem to another. In the anti-air defense task, the relevant factors are the damage that a missile can cause and the cost of intercepting one. The number of influencing factors (*N*) is also subject to change. As a result the weight vector learned in one task may be completely useless in another, even when modeling the same player. In a video game, this would necessitate learning a new weight vector in every new team-task involving the same set of agents/players. This is surely wasteful and time-consuming, and a major reason behind the impracticality of machine learning in games.

We argue that there are some characteristics that are intrinsic to a player and do not change from one task to another. E.g., his risk-sensitivity, team-spirit and other similar traits can be considered to be a constant. A modeler might consider a fixed, given set of traits $T = \{t_1, t_2, ..., t_S\}$ that reasonably capture any player's disposition. The way to choose the number of traits and their actual meaning is a design issue. The relative importance of these traits (let's say p_j for t_j, where we assume that the t_j's are given but the p_j's are the ones that has to be learned) to a player may also be considered fixed (but unknown) for a given genre of tasks, such as tasks in an RTS game. However, the effect of these traits on the decision factors (distance, quadrant etc) can vary from one task to another. For instance, team spirit can prompt a player to try to increase quadrant value[1] (in predator-prey task), while it may prompt the players to decrease damage value (in anti-air defense task), since damage occurs to shared properties. We model the influence factor between the j^{th} trait (t_j) and the k^{th} decision factor (X_k) as a three-valued (-1, 0 or 1) variable, f_k^j because we are only interested in capturing whether the j^{th} trait increases, decreases or leaves unchanged the decision variable X_k. Combined with the importance of a trait (p_j for t_j), we can directly relate to the weight of a decision variable using the following equation:

$$w_k = \sum_{j=1}^{j=S} p_j f_k^j \qquad (3)$$

Substituting this in Equation (1), we get the action decision of the player in terms of his traits. This reduces the learning task from acquiring the (old) weights w_k, to the (new) weights p_j. The major gain of this decomposition is that the weights to be learned are no longer associated with task specific factors (such as distance, quadrant in predator-prey vs. damage, cost in anti-air defense). Instead, they are now associated with the traits of a player that are fixed over the entire genre of tasks. Note from equation (3) that the right-hand-side does still include the task-specific factor f_k^j, but this can now be easily determined from the task (one of three possible values) and the semantics of the decision factors. For instance, team-spirit should *increase* quadrant value in predator-prey, but *decrease* damage value in anti-air defense. As a result, if we can acquire p_j in one task,

[1] The quadrant value is high if players are spread out in different quadrants, but low if they are concentrated on one or a few, since this makes it harder to catch the prey.

we can reuse this value without learning it again in a new task. Now Equation (3) will let us choose the values of f_k^j in a way that the weight w_k of a certain trait would make sense with the problem that we're trying to solve.

It may seem at this point, that the above formulation of the learning problem makes learning unnecessary after the first task. After all, if we can acquire all the trait weights ($p_j, j = 1,..., S$) in one task, what remains to be learned in the next task? There are at least two reasons why this is not the case. Firstly, all trait weights may not be learnable in any one given task, because some f_k^j may be 0. If f_k^j is 0 for all *k*, then p_j is basically irrelevant to the current task, and hence cannot be learned in the current task. In other words, a task may only invoke a proper subset of the set of S traits designed beforehand; so the rest cannot be learned in that task. The second reason is more subtle and will be discussed in the next subsection, after we have introduced the new weight update rule.

The new learning rule

The new weight learning rule for our formulation can be derived in the same way as in Suryadi and Gmytrasiewicz (1999). Once again, gradient descent on the error function $E(t)$ gives us the necessary change to the new weight p_j by noting that

$$\frac{\partial e_i(t)}{\partial p_j} = \sum_{k=1}^{k=N} f_k^j \aleph_k^i(t) .$$

Plugging this value in Equation (2) in place of $\frac{\partial e_i(t)}{\partial w_k(t)}$, we get

$$\Delta p_j = \eta \sum_{i=1}^{i=M} e_i(t) \sum_{k=1}^{k=N} f_k^j \aleph_k^i(t) \qquad (4)$$

This is the new weight update rule.

Now it is possible that f_k^{j1} and f_k^{j2} are the same for all decision variables X_k, since they have a very limited set of possible values. Then, by Equation (4),

$$\Delta p_{j1} = \Delta p_{j2} .$$

This constrains the joint dynamics of (p_{j1}, p_{j2}) such that they may converge to values that are completely different from their target values. In particular, they will converge to identical values if their initial values are identical, even though their target values may be different. Nevertheless, Equation (3) will be preserved and p_j's will still give the correct target values of w_k. However, since convergence to the target p_j cannot be guaranteed, the values learned in one task may not be accurate for another task. *We claim that these inaccurate values are still a better point for initialization in the next task, than simple default initialization.* We verify this intuition empirically.

EXPERIMENTAL RESULTS

We have used two different tasks for testing the relative efficiency of our approach, compared to Suryadi and Gmytrasiewicz (1999). These are the anti-air defense task, and the predator-prey task, described below. The learning agent works in a team with exactly one other player in both of these tasks. Target weights that simulate the player's decision are chosen, but are not made known to the agent. The agent must learn these weights from repeated joint interaction with the environment, and the resulting observations of the player's action choices. The overall plan

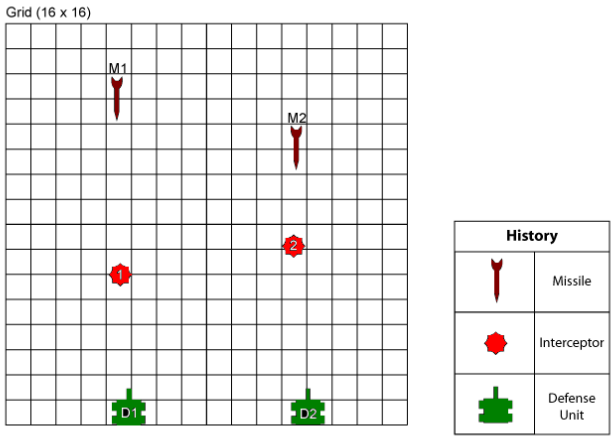

Figure 4: The Anti-air defense task

is to make the agent learn some trait weights in the anti-air defense task, and use these for initialization in the predator-prey task. The trait weights that were not learned in the former, can be initialized to some default values in the latter. We then compare the learning rate with the baseline method (from Suryadi and Gmytrasiewicz (1999)) which must learn from scratch the player model in the latter task. We will consider our knowledge-transfer approach a success if the learning rate is superior to the baseline method. Two particular hallmarks of this superiority are

- a difference in the initial performance; our approach should have a lower initial error by virtue of informed initialization
- a difference in the convergence value; our approach should have lower error on convergence, i.e., it should learn a better model faster than in Suryadi and Gmytrasiewicz (1999).

Anti-air Defense task

This is a 16×16 grid-world adopted from Suryadi and Gmytrasiewicz (1999), with two agents, D1 and D2, the learner and the human player, as shown in Figure 4. In every round of the game, two missiles, M1 and M2 are fired and

the agents must shoot down these missiles with interceptors. If missed, a missile will cause damage proportional to its size. An agent incurs a cost proportional to the accuracy and efficiency with which he intercepts a missile. Without communicating, the agents must choose different missiles to intercept, lest one of the missiles makes it through. Hence, there are two actions available to each agent (M1 or M2) and two decision variables, X_1 = Damage, and X_2 = Cost.

Predator-prey task

This is an 8×8 grid-world with two predators, the learner and the human player, shown in Figure 5. There is a prey that executes a random walk and the predators must catch the prey by either reaching grid-cells neighboring the preys that are on opposite quadrants relative to the prey, or by cornering it. Each agent can select one of 5 actions, viz., go north, west, south, east or stay put. Again for simplicity, there are just 2 decision variables, X_1 = Distance, and X_2 = Quadrant. Distance is the distance between the prey and player if the player executes a chosen action. Quadrant is a score with 3 possible values, viz., 0 (if both predators are on the same quadrant relative to the prey if the player executes the chosen action), 1 (if they are on different quadrants but same side/half of the prey) and 2 (if they are on diametrically opposite quadrants relative to the prey). Therefore, higher values of Quadrant and lower values of Distance are more conducive to catching the prey. In order to surround the prey the learner needs to correctly predict the player's next move.

Experiments and analysis

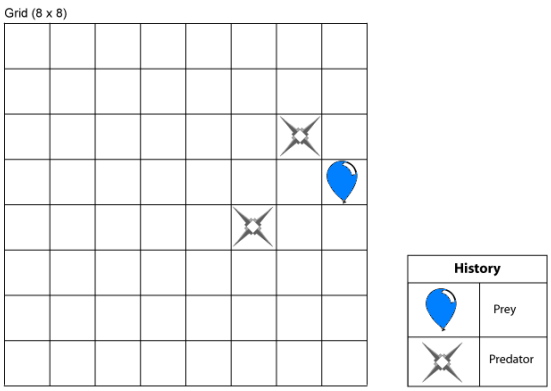

Figure 5: The Predator-prey task

For simplicity, we assume a small set of only 3 traits. To simulate the player's decisions without the involvement of a real human player, we used the values $[p_1, p_2, p_3] = [0.95, 0.272, 0.13]$ which is a normalized vector. Since we assume that the *p*-values are available (albeit not to the modeler), the semantics of the traits are immaterial in this paper; so we leave the traits unnamed. We choose these (and *f*) values such that the resulting weights (from equation (3)) make sense with respect to the features X_k. Considering a specific set of named traits will be necessary in actual human trials where the *p*-values will not be available for experimental validation.

In the anti-air defense task, we used $[f_1^1, f_1^2, f_1^3] = [-1, 0, 0]$ for X_1 (i.e., Damage) and $[f_2^1, f_2^2, f_2^3] = [0, 0, 1]$ for X_2 (i.e., Cost). This effectively excludes t_2 from this task, so that p_2 is impossible to learn here. Our simulations show that the agent is able to learn p_1 and p_3 to a reasonable accuracy (0.98 and 0.18 respectively, after 1000 iterations with $\eta = 0.5$), but p_2 stays at the default initial value[2]. This gives the learner a partial picture of the player's disposition that it then leverages in the next task.

In the predator-prey task, the learner initializes p_1 and p_3 to the values it had learned in the previous task, but initializes p_2 to 0 (the default value, since it was not learned). The influence factors in this task were chosen as $[f_1^1, f_1^2, f_1^3] = [-1, 1, 1]$ for X_1 (i.e., Distance) and $[f_2^1, f_2^2, f_2^3] = [0, 1, 1]$ for X_2 (i.e., Quadrant). Thus all three traits are needed in this task. The agent then learns all 3 trait-weights (using Equation 4) over 1000 iterations, making a prediction in each iteration based on the current weights and tallying it with the observed choice of the player. A count of the errors in prediction is kept and a cumulative average of these errors is plotted in Figure 6 (titled "with transfer"), averaging over 3 runs.

Additionally, we let the learner use the baseline approach from Suryadi and Gmytrasiewicz (1999) (Equation (2)) to learn w_1 and w_2 from repeated observations of the player's action choices in predator-prey, starting from default weights of 0, since this method does not allow for knowledge transfer. The cumulative average of the resulting number of errors is also averaged over 3 runs and plotted in Figure 6 (titled "without transfer"). A comparison of the two plots in Figure 4 shows that

- the initial error is lower with our approach
- the convergence value of the error is lower with our approach.

Thus we have met both criteria of successful transfer with the proposed technique.

[2] Since the *p*-values are limited in the range [-1,1], we use the default initial value of 0.

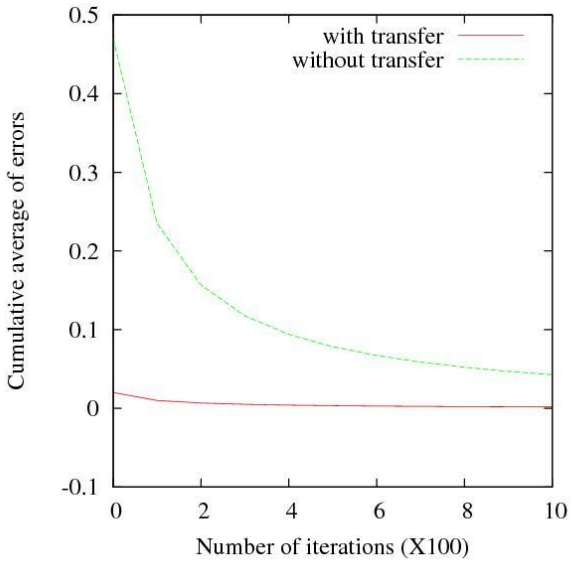

Figure 6: Difference between Agent Modeling with knowledge transfer and without transfer.

CONCLUSIONS

We have presented a reusable player modeling scheme that exploits knowledge (partial model) acquired in previous tasks to bootstrap learning in subsequent tasks. While tasks can be markedly different, the similarity that knowledge transfer exploits is in the fact that the model is of the *same* player that the agent repeatedly meets in a series of tasks. We have shown experiments in two simple tasks that demonstrate the advantage of our approach compared to previous work.

We treat the results in this paper as a proof of concept, and plan a more elaborate study of our approach in more complex RTS games. Additionally, we plan to incorporate actual human decisions instead of simulated decisions.

REFERENCES

Abdi, H., Valentin, D., & Edelman, B. (1999). Neural networks. Thousand Oaks (CA): Sage

Banerjee B., and Stone P. 2007. General Game Learning using Knowledge Transfer. *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, Hyderabad, India, January 6-12.

Carmel D. and Markovitch S. 1996. Learning models of intelligent agents. *AAAI/IAAI*, 1:62–67.

Charniak E. 1991. Bayesian networks without tears: making Bayesian networks more accessible to the probabilistically unsophisticated. *AI Mag. Volume 12 issue 4, p50-63. AAAI*, Menlo Park, CA, USA

Howard R. A. and Matheson J. E. 1984. Influence diagrams (article dated 1981). Howard, R.A. and Matheson, J.E. (Eds.), Readings on the principles and applications of decision analysis, 2:719–762.

Koller, Daphne & Milch, Brian, 2003. "Multi-agent influence diagrams for representing and solving games," Games and Economic Behavior, Elsevier, vol. 45(1), pages 181-221, October.

Konidaris G. and Barto A. 2007. Building Portable Options: Skill Transfer in Reinforcement Learning. *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, Hyderabad, India, January 6-12.

Roy D. M. and Kaelbling L. P. 2007. Efficient Bayesian Task-Level Transfer Learning. *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, Hyderabad, India.

Russell S. J. and Norvig P. 1995. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ.

Pearl J. 1988. Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kauffman, San Mateo, CA.

Suryadi D. and Gmytrasiewiez P. J. 1999. Learning models of other agents using influence diagrams. *UM '99: Proceedings of the seventh international conference on User modeling*. Pages: 223-232.

Sutton, R. S., Precup, D., & Singh, S. 1999. *Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning*. Artificial Intelligence, 112, 181--211

Widrow, B., and Hoff Jr., M. E. 1960. Adaptive switching circuits. IRE WESCON Convention Record 96–104