systems[2].

Prior to conducting the research reported in this chapter, we used neural networks to teach an agent to shoot a moving ball into a goal in a simulator based closely upon the Dynasim system [Sahota 96]. The simulator models the non-holonomic wheeled Dynamite robots [Sahota *et al.* 95]. In this scenario, we were able to use one neural network to time the agent's approach towards the ball, and another to aim the ball. The learned behavior successfully enabled the agent to redirect a moving ball with varying speeds and trajectories into specific parts of the goal. By carefully choosing the input representation to the neural networks so that they would generalize as much as possible, the agent was able to use the learned behavior in all quadrants of the field even though it was trained in a single quadrant [Stone and Veloso 98d].

Learning was applied in both simulated and real robotic, to hit a stationary ball into the goal using the concept of learning from easy missions [Asada *et al.* 94a]. By first placing the ball directly in front of the goal and then gradually moving it farther away, thus making the task harder, the learning time was reduced from exponential to linear in the size of the state space within an RL framework. The robots achieved 70% success in simulation and 40% success in the real world.

Another early example of learning an individual skill in the RoboCup soccer server involved using a neural network to enable an agent to learn when to shoot and when to pass [Matsubara *et al.* 96]. An agent was given the ball near the opponent's goal with a goaltender blocking the way and a teammate nearby. Based on the positions of the ball, the goaltender, and the teammate, the agent with the ball learned when it was best to shoot directly at the goal and when it was best to pass.

In a similar setup, using the same Dynasim-based simulator mentioned above, we used memory-based learning to allow an agent to learn when to shoot and when to pass the ball [Stone and Veloso 96a]. This learning was of an individual skill since the goaltender and the teammate used deterministic, fixed behaviors.

[2] A full discussion of related robotic soccer systems, including more recent learning approaches, appears in Chapter 9 (Section 9.2).

Chapter 7

Learning a Team Behavior

This chapter serves a dual purpose. It presents the third and final implemented learned layer in our simulated robotic soccer layered learning implementation, and it simultaneously introduces a separate contribution of this thesis, namely the team-partitioned, opaque-transition reinforcement learning (TPOT-RL) algorithm [Stone and Veloso 99c]. TPOT-RL is a new machine learning method used to train collaborative and adversarial team behaviors. In the robotic soccer context, we use TPOT-RL to learn pass selection, taking advantage of the learned pass-evaluation capability described in Chapter 6 to construct the input representation for learning.

This chapter is organized as follows. In Section 7.1, I introduce the new collaborative and adversarial robotic soccer behavior and motivate the need for a new multi-agent learning algorithm in order to train it. Section 7.2 formalizes the TPOT-RL algorithm in domain-independent terms and Section 7.3 applies it to simulated robotic soccer with extensive empirical testing. In Section 7.4, TPOT-RL is applied in a different multi-agent domain—network routing—in order to verify the generality of the new algorithm. Sections 7.5 and 7.6 are devoted to discussion and related work respectively.

7.1 Motivation

In this section, I motivate the need for a new RL algorithm in order to continue our layered learning implementation in the simulated robotic soccer domain. I do so by first describing the task to be learned (Section 7.1.1); then showing that existing RL algorithms cannot be used for this task (Section 7.1.2). I formally define the TPOT-RL algorithm in Section 7.2.

7.1.1 Pass Selection

Once young soccer players have learned how to judge whether a particular pass will succeed, they are ready to learn how to act strategically as part of a team. Based on experience, they can start to estimate the long-term effects of their actions within the context of a game in which the true goal is to help one's team to beat the opponents. Similarly, our simulated robotic soccer agents cooperate to achieve the team's goal of beating the opponents by learning pass selection, a team behavior. Good pass selection requires an understanding of the long-term effects of local decisions given the behaviors and abilities of teammates and opponents.

In Section 6.3.3, I presented a full team behavior based upon the first two learned layers of our layered learning implementation; i.e. ball interception and pass evaluation. Specifically, the learned pass evaluation function was incorporated into the DT (decision tree) receiver choice function (RCF), that determines to which teammate an agent should pass when it has possession of the ball. In the DT RCF, the learned pass-evaluation capability is used in a heuristic, hand-coded function: the potential receivers are limited to those that are at least as close to the opponent's goal as the agent with the ball (C_r), and the agent always passes the ball to the potential receiver to which it can pass with the highest confidence of success according to the trained DT (maximum Φ(passer,receiver)).

While the DT RCF was useful for verifying that the pass-evaluation capability can be used in full-game situations, in reality the choice of where to pass is much more complicated than allowed for by the DT RCF. For one thing, there may be situations in which the best pass is to a receiver that is farther away from the goal than the passer. For another, the receiver that is most likely to successfully *receive* the pass may not be the one that will subsequently act most favorably for the team.

Figure 7.1 illustrates a situation in which the player with the ball, agent b, may want to pass backwards (to the left in the figure). Agent b and its teammates are attacking the right goal. The white arrows emanating from the ball indicate potential passes and their labels indicate the predictions by the DT as to whether or not they will succeed: "S" for success and "F" for failure. In this situation, the DT predicts that all three possible forward passes, to teammates f, g, and h, will fail. However, there are two predicted-successful backwards passes to teammates c and d. If agent b's goal is simply to complete a pass to a teammate, it should clearly pass to teammate c or d.

However, in the context of the team's high-level task of kicking the ball into the opponent's goal, the choice is not so clear:

- Suppose that the opponents are very bad at intercepting the ball. Then the pass to teammate g may be best for the team. Recall that the decision tree is trained off-line in an artificial situation and does not adjust its predictions based on the current opponent. Thus despite the DT's prediction, teammate g may be able to receive the pass. Since it is closest to opponent's goal, the team might be best off if agent b passes *forwards* to teammate g.

- Suppose instead that the opponents are good at intercepting the ball but that teammate c has a poor decision-making policy. Despite the appearance that its best option would be to pass to teammate f, agent c instead decides to pass to teammate e, which would likely result in the opponents stealing the ball. Again, the team might be better off if agent b passes *forwards* to teammate f, g, or h despite the DT's prediction.

- On the other hand, suppose that the opponents are good at intercepting the ball and that teammate c has a good decision-making policy: it would pass to teammate f after receiving the ball. Then the team will probably be best off in the long run if agent b passes passes *backwards*, away from the goal, to teammate c.

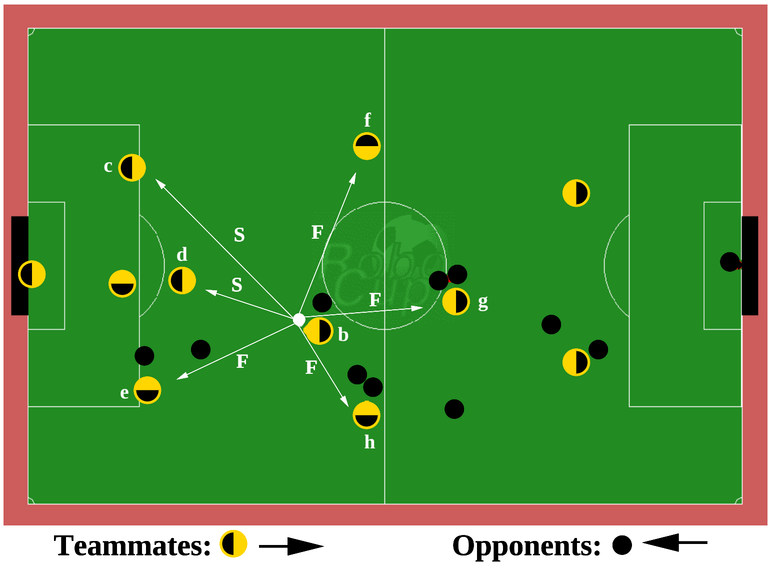

Figure 7.1: An example of when it might be useful to pass backwards. The white arrows emanating from the ball indicate potential passes and their labels indicate predictions by the trained DT as to whether or not they will succeed: "S" for success and "F" for failure. The teammates are attacking the right goal; the opponents, the left.

In short, when learning an RCF, the local pass-evaluation information, while helpful, is not enough on which to base a decision. The agent needs to learn *strategic* information which depends on the behaviors of teammates and opponents and which can only be measured by its team's long-term success at achieving its collective goal against an opponent team in a real game. Therefore, as opposed to both lower-level learned behaviors (ball interception and pass evaluation), the pass-selection behavior must be trained *on-line* in a game situation against a particular opponent.

When choosing an ML method for learning an RCF, there are five other task characteristics that are important.

- The pass-selection decision depends on a *huge amount of information*, including the agent's current location on the field, the current locations of all the teammates and opponents, the teammates' abilities to receive a pass, the opponents' abilities to intercept passes, teammates' subsequent decision-making capabilities, and the possibly-changing opponents' strategies. As presented in Section 2.2, the simulated robotic soccer domain has more than 10^{198} states.

- The agents must learn with *limited training examples*. When running with two full teams of agents, the soccer server is computationally intensive. To run at full speed, the server and the clients must be distributed across at least two 266 MHz computers. And in real games, empirically each agent gets the ball, and therefore the opportunity to collect a training example, about once a minute on average. With additional computational power, the simulation can be sped up somewhat. But unlike most simulations, and more like real robotic domains, training data is relatively difficult and time-consuming to collect.

- The merit of a particular decision can only be measured by the *long-term* performance of the team as a whole and thus only becomes clear over time.
- Since the expected reward for taking a particular action depends on teammates' behaviors, this expected reward changes as teammates simultaneously learn to improve their behaviors. In ML terms, the task to be learned is said to be a *shifting concept*.
- As the teammates each stay in a different geographical region of the field (their positions within the team formation—see Chapter 3), the learning task is *partitioned among the teammates*: each agent only learns how to act when located in a specific part of the field.

In addition to the above task characteristics, recall one important domain characteristic from the presentation of simulated robotic soccer in Chapter 2 that has a bearing on the choice of ML algorithm. In simulated robotic soccer there are *opaque transitions*: when an agent takes an action (passes the ball), the resulting state transition as well as the subsequent actions taken by other agents—both teammates and adversaries—and their resulting state transitions are often unknown.

In summary, the characteristics of an ML algorithm needed for learning pass selection are:

1. on-line;
2. capable of dealing with a large state space despite limited training;
3. capable of learning based on long-term, delayed reward;
4. capable of dealing with shifting concepts;
5. works in a team-partitioned scenario; and
6. capable of dealing with opaque transitions.

7.1.2 Reinforcement Learning for Pass Selection

Although no previously existing ML algorithm has all of the above characteristics, some of the ones that come closest are reinforcement learning (RL) algorithms. RL is an effective paradigm for training an agent to act in its environment in pursuit of a goal. RL techniques rely on the premise that an agent's action *policy* affects its overall reward over time. A policy is defined as an agent's mapping from the state it is in to the action it will take. In the robotic soccer context, an RCF can be thought of as a policy over the set of states in which an agent has possession of the ball. Throughout this chapter, the RCF is the only part of the agents' policies being learned or otherwise varied in any way.

As surveyed in [Kaelbling *et al.* 96], standard RL algorithms are on-line learning algorithms that can learn control policies for Markov decision tasks based on long-term, delayed reward. Whether model-based (e.g. prioritized sweeping [Moore and Atkeson 93]) or model-free (e.g. TD(λ) [Sutton 88] and Q-learning [Watkins 89]), these RL techniques rely on a single agent being able to observe the environmental state transitions in order to propagate reward back from states that generate reward to previous states. In addition, these techniques require time that is polynomial in the number of states. This "curse of dimensionality" typically limits them to relatively small state spaces with abundant training opportunities[1].

In contrast, motivated by the presentation in Section 7.1.1, we restrict our attention to domains with the following characteristics:

- There are multiple agents organized in a team: they have a common long-term goal.
- There are opaque state transitions: agents cannot observe state transitions when they or other agents act and therefore cannot build a model of transitions in the domain.
- There are too many states and/or not enough training examples for traditional RL techniques to work.
- The target concept can change during the course of learning, for example as a result of other agents in the environment changing their policies. Thus, the Markov property, that the effect of an action depends only on the current state of the world, does not hold.
- There is long-range reward available: agents can notice the long-term effects of their actions by directly observing the environment.
- There are *action-dependent features* available. By "action-dependent," we mean that the feature value depends both on the current world state *and* the action being considered: in a given state, each possible action has its own action-dependent classification.

  As opposed to the ultimate learning goal of determining the long-term reward to be expected when executing an action, action-dependent features classify the predicted short-term effects of actions based on local state information.

In order to enable teams of agents to learn in such domains, we introduce team-partitioned, opaque-transition reinforcement learning (TPOT-RL). Like previous RL methods, TPOT-RL allows agents to learn on-line and from delayed rewards. TPOT-RL extends RL to work in domains with the above characteristics.

7.2 TPOT-RL

This section presents TPOT-RL in detail. Like Q-learning, TPOT-RL learns a *value function* that maps state-action pairs to expected rewards. TPOT-RL includes three main adaptations to the standard RL paradigm:

- The value function is partitioned among the team, with each agent only learning for states from which it can act. All agents are trained simultaneously with a gradually decreasing exploration rate and an increasing exploitation rate.

[1] One notable exception is TD-Gammon [Tesauro 94], which achieved champion status in backgammon using RL. It dealt successfully with backgammon's large state space by introducing hand-crafted features and allowing for an extremely large amount of training.

- Action-dependent features are used to produce an aggressively generalized feature space, which is used as the input representation for learning. While other RL approaches aggregate states to reduce the size of the learning task (e.g. [McCallum 96]), action-dependent features enable the creation of a particularly small but informative feature space for learning.

- Long-term, discounted rewards are gathered directly from the rewarding states in the environment rather than being propagated back through intermediate states, as the state-transition probabilities are not known. While other RL approaches have learned based on reward from eventual rewarding states (e.g. TD(1) [Sutton 88]), none has been applied in a multi-agent scenario and they generally do not discount reward based on the time between an action and the eventual reward.

Formally, a policy is a mapping from a state space S to an action space A such that the agent using that policy executes action a whenever in state s. At the coarsest level, when in state s, an agent compares the expected, long-term rewards for taking each action $a \in A$ and chooses an action based on these expected rewards. These expected rewards are learned through experience.

Designed to work in real-world domains with far too many states to handle individually, TPOT-RL exploits action-dependent features to create a small feature space V. V is used as a component of the input representation of the learned value function $Q : V \times A \mapsto \mathbb{R}$.

In short, the policy's mapping from S to A in TPOT-RL can be thought of as a 3-step process:

State generalization: The state s is generalized to a feature vector v using the state generalization function $f : S \mapsto V$.

Value function learning: The feature vector v is used as an input to the learned value function $Q : V \times A \mapsto \mathbb{R}$, which estimates the expected reward for taking each possible action.

Action selection: An action a is chosen for execution and its long-term, observed reward is used to further update Q.

While these steps are common in other RL paradigms, each step has unique characteristics in TPOT-RL. State generalization, value function learning, and action selection in TPOT-RL are further specified in Sections 7.2.1, 7.2.2, and 7.2.3 respectively.

7.2.1 State Generalization

$f : S \mapsto V$ maps the current state of the world, s, to the feature vector used for learning, v. f relies on a unique approach to constructing V. Rather than discretizing the various dimensions of S, it uses an *action-dependent* feature function.

The action-dependent feature function

$$e : S \times A \mapsto U$$

evaluates each possible action $a_i \in A$ based on s. U is a discrete set of features reflecting expected short-term effects of actions. Unlike Q, e does not produce the expected long-term reward of taking an action; rather, it classifies the likely short-term effects of the action. For example, since state transition probabilities are unknown, e might predict the transition to be caused by executing action a_i based on s.

In the multi-agent scenario, other than one output of e for each action, the feature space V also involves one coarse component that partitions the state space S among the agents. The partition function

$$P : S \mapsto M$$

breaks the state space into $|M|$ disjoint partitions to be divided among the teammates, with $|M| \geq m$ where m is the number of agents in the team. In particular, if the set of possible actions in state s is $A = \{a_0, a_1, \ldots, a_{n-1}\}$, then

$$\begin{aligned} f(s) &= \langle e(s, a_0), e(s, a_1), \ldots, e(s, a_{n-1}), P(s) \rangle, \text{ and so} \\ V &= U^{|A|} \times M. \end{aligned}$$

Thus, $|V| = |U|^{|A|} * |M|$. Since the goal of constructing V is to create a feature space that is smaller than the original state space, the ranges of the action-dependent feature function and partition function, U and M respectively, are ideally as small as they can be without abstracting away the useful information for learning.

This state generalization process reduces the complexity of the learning task by constructing a small feature space V which partitions S into $|M|$ regions. Each agent need learn how to act only within its own partition(s). Nevertheless, for large sets A, the feature space can still be too large for learning, especially with limited training examples. Our particular action-dependent formulation allows us to reduce the effective size of the feature space in the value-function-learning step. Choosing features for state generalization is generally a hard problem. While TPOT-RL does not specify the function e, our work illustrates effective choices of e.

7.2.2 Value Function Learning

As we have seen, TPOT-RL uses action-dependent features. When using action-dependent features, we can assume (heuristically) that the expected long-term reward for taking action a_i depends only on the feature value related to action a_i. That is,

$$Q(f(s), a_i) = Q(f(s'), a_i) \tag{7.1}$$

whenever $e(s, a_i) = e(s', a_i)$ and $P(s) = P(s')$. Recall that

$$\begin{aligned} f(s) &= \langle e(s, a_1),\ \ldots, e(s, a_{n-1}),\ P(s) \rangle \\ f(s') &= \langle e(s', a_1), \ldots, e(s', a_{n-1}), P(s') \rangle \end{aligned}$$

Another way of stating this same assumption is that $Q(f(s), a_i)$ depends upon $e(s, a_i)$ and is independent of $e(s, a_j)$ for all $j \neq i$.

Without this assumption, since there are $|A|$ actions possible for each element in V, the value function Q has $|V| * |A| = |U|^{|A|} * |M| * |A|$ independent values. Under this assumption, however, the Q-table has at most $|U|^1 * |M| * |A|$ entries: for each action possible from state s, only one of the $|A|$ action-dependent feature values $e(s, a_i)$ comprising $f(s)$ is relevant. Therefore, even with only a small number of training examples available, we can treat the value function Q as a lookup-table without the need for any complex function approximation. To be precise, Q stores one value for every possible combination of $a \in A$, $e(s, a) \in U$, and $P(s) \in M$.

For example, Table 7.1 shows the entire feature space for one agent's partition of the state space when $|U| = 3$ and $|A| = 2$. $U = \{u_0, u_1, u_2\}$, $A = \{a_0, a_1\}$. $q_{i,j}$ is the estimated value of taking action a_i when $e(s, a_i) = u_j$. Since this table is for a single agent, $P(s)$ remains constant.

There are $|U|^{|A|} = 3^2$ different entries in feature space with $|A| = 2$ Q-values for each entry: one for each possible action. $|U|^{|A|} * |M|$ is much smaller than the original state space for any realistic problem, but it can grow large quickly, particularly as $|A|$ increases. However, as emphasized by the right side of Table 7.1—which is simply a condensed representation of the left side—under the assumption described above, there are only $3 * 2$ independent Q-values to learn, reducing the number of free variables in the learning problem from 18 to 6, or by 67%, in this case.

$e(s,a_0)$	$e(s,a_1)$	$Q(f(s),a_0)$	$Q(f(s),a_1)$
u_0	u_0	$q_{0,0}$	$q_{1,0}$
u_0	u_1	$q_{0,0}$	$q_{1,1}$
u_0	u_2	$q_{0,0}$	$q_{1,2}$
u_1	u_0	$q_{0,1}$	$q_{1,0}$
u_1	u_1	$q_{0,1}$	$q_{1,1}$
u_1	u_2	$q_{0,1}$	$q_{1,2}$
u_2	u_0	$q_{0,2}$	$q_{1,0}$
u_2	u_1	$q_{0,2}$	$q_{1,1}$
u_2	u_2	$q_{0,2}$	$q_{1,2}$

$\Longrightarrow$

$e(s,a_i)$	$Q(f(s),a_0)$	$Q(f(s),a_1)$
u_0	$q_{0,0}$	$q_{1,0}$
u_1	$q_{0,1}$	$q_{1,1}$
u_2	$q_{0,2}$	$q_{1,2}$

Table 7.1: A sample Q-table for a single agent when $|U| = 3$ and $|A| = 2$: $U = \{u_0, u_1, u_2\}$, $A = \{a_0, a_1\}$. $q_{i,j}$ is the estimated value of taking action a_i when $e(s, a_i) = u_j$. Since this table is for a single agent, $P(s)$ remains constant. The table on the right emphasizes that there are only 6 independent q values in the table on the left.

The Q-values learned depend on the agent's past experiences in the domain. In particular, after taking an action a while in state s with $f(s) = v$, an agent receives reward r and uses it to update $Q(v, a)$ from its previous value $Q(v, a)$ as follows:

$$Q(v, a) = Q(v, a) + \alpha(r - Q(v, a)) \tag{7.2}$$

where α is the learning rate. Since state transitions are opaque, the agent cannot use the dynamic programming (and Q-learning) approach of updating the value function based on the value of the state that results from executing action a. Instead, the reward r is derived from the observable environmental characteristics—those that are captured in S—over a maximum number of time steps t_{lim} after the action is taken. The reward function

$$R : S^{t_{lim}} \mapsto \mathbb{R}$$

returns a value at some time no further than t_{lim} in the future. The reward is discounted based on the amount of time between acting and receiving the reward. During that time, other teammates (or opponents, if any) can act in the environment and affect the action's outcome, but the agent may not be able to observe these actions. In practice, the range of R is $[-Q_{max}, Q_{max}]$ where Q_{max} is the reward for immediate goal achievement. In order to associate the eventual reward with the action it took, the agent must keep track of the last action taken a_i and the feature vector v at that time.

Notice from Equation 7.2 that in TPOT-RL, the updated action value depends only on the previously stored action value in the same state as opposed to chains of learned state values. That is, when updating $Q(v, a)$ TPOT-RL doe not reference any $Q(v', a')$ such that $v \neq v'$ or $a \neq a'$. If the actual value of executing action a from state s with $f(s) = v$ changes, the agent can adjust by simply executing a in such a state several times: no other states-action pairs need to be updated first. Therefore, TPOT-RL is able to adopt relatively quickly to shifting concepts.

The reward function, including t_{lim} and Q_{max}, is domain-dependent. One possible type of reward function is based entirely upon reaching the ultimate goal. In this case, an agent charts the actual (long-term) results of its policy in the environment. However if goal achievement is infrequent, a reward function based on *intermediate reinforcement*, which provides feedback based on intermediate states towards the goal, may be needed.

7.2.3 Action Selection

As in all RL techniques, the issue of exploration vs. exploitation is important for TPOT-RL. Particularly since the target concept can shift due to teammates learning and changing their policies, or due to changes in policies of opponents (if any), it is important for agents to gather information about the value of actions that are currently considered sub-optimal by the value function. Any standard exploration heuristic, such as the randomized Boltzmann exploration strategy [Kaelbling *et al.* 96], could be used.

Informative action-dependent features can be used to reduce the free variables in the learning task still further at the action-selection stage if the features themselves discriminate situations in which actions should not be used. For example, suppose we can define a set $W \subseteq U$ such that if $e(s, a) \notin W$, then a should not be considered as a potential action from state s.

Formally, consider $W \subseteq U$ and $B(s) \subseteq A$ with $B(s) = \{a \in A | e(s, a) \in W\}$. When in state s, the agent then chooses an action from $B(s)$ instead of from the entire action set A, either randomly when exploring or according to maximum Q-value when exploiting. In effect, W acts in TPOT-RL as an action filter which reduces the number of options under consideration at any given time. Again, any standard exploration heuristic can be used over the possible actions in $B(s)$. Of course, exploration at the filter level can also be achieved by dynamically adjusting W.

For example, Table 7.2, illustrates the effect of varying $|W|$. Notice that when $W \neq U$, it is possible that $B(s) = \emptyset$: $\forall a_i \in A, e(s, a_i) \notin W$. In this case, either a random action can be chosen, or rough Q-value estimates can be stored using sparse training data. This condition becomes rarer as $|A|$ increases. For example, with $|U| = 3$, $|W| = 1$, and $|A| = 2$ as in Table 7.2(b), $4/9 = 44.4\%$ of feature vectors have no action that passes the W filter. However, with $|U| = 3$, $|W| = 1$, and $|A| = 8$ only $256/6561 = 3.9\%$ of feature vectors have no action that passes the W filter. If $|U| = 3$, $|W| = 2$ and $|A| = 8$, only 1 of 6561 feature vectors fails to pass the filter. Thus using W to filter action selection can reduce the number of free variables in the learning problem without significantly reducing the coverage of the learned Q-table. However, there is always the danger that the best possible action from a particular state could be filtered out: an informed, heuristic choice of W is required.

$e(s,a_0)$	$e(s,a_1)$	$Q(f(s),a_0)$	$Q(f(s),a_1)$
u_0	u_0	$q_{0,0}$	$q_{1,0}$
u_0	u_1	$q_{0,0}$	—
u_0	u_2	$q_{0,0}$	$q_{1,2}$
u_1	u_0	—	$q_{1,0}$
u_1	u_1	—	—
u_1	u_2	—	$q_{1,2}$
u_2	u_0	$q_{0,2}$	$q_{1,0}$
u_2	u_1	$q_{0,2}$	—
u_2	u_2	$q_{0,2}$	$q_{1,2}$

(a)

$e(s,a_0)$	$e(s,a_1)$	$Q(f(s),a_0)$	$Q(f(s),a_1)$
u_0	u_0	—	—
u_0	u_1	—	—
u_0	u_2	—	$q_{1,2}$
u_1	u_0	—	—
u_1	u_1	—	—
u_1	u_2	—	$q_{1,2}$
u_2	u_0	$q_{0,2}$	—
u_2	u_1	$q_{0,2}$	—
u_2	u_2	$q_{0,2}$	$q_{1,2}$

(b)

Table 7.2: The resulting Q-tables when $U = \{u_0, u_1, u_2\}$, $A = \{a_0, a_1\}$, and (a) $W = \{u_0, u_2\}$, or (b) $W = \{u_2\}$.

7.2.4 Summary

By partitioning the state space among teammates, by using action-dependent features to create a coarse feature space and to filter actions, and with the help of a reward function based entirely on individual observation of the environment, TPOT-RL enables team learning in complex multi-agent, non-stationary environments even when agents cannot track state transitions.

In order to apply TPOT-RL to particular learning problems, as we do in Sections 7.3 and 7.4, the following functions and variables must be specified within the domain:

- The action-dependent feature function e and its range U.
- The partition function P and its range M.
- The reward function R including variables Q_{max} and t_{lim}.
- The learning rate α.
- The action-filtering set $W \subseteq U$.

In the two TPOT-RL implementations reported in Sections 7.3 and 7.4, the specifications of these variables are all indented for emphasis.

7.3 TPOT-RL Applied to Simulated Robotic Soccer

In this section, I describe our application of TPOT-RL to a complex multi-agent learning task, namely pass selection in simulated robotic soccer. Pass selection is the third layer in our layered learning implementation.

In the soccer domain, we apply TPOT-RL to enable each teammate to simultaneously learn a high-level action policy, or receiver choice function (RCF). The RCF is a function that determines what an agent should do when it has the opportunity to kick the ball. When it does not have the ball, the agent acts according to a manually created behavior as defined in Section 6.3.3.

As presented in Section 6.3.1, the input of the RCF is the agent's perception of the current world state; the output is a target destination for the ball in terms of a location on the field, e.g. the opponent's goal. In our experiments reported in this section, each agent has 8 possible actions in A: it can pass towards either of the two goals, towards any of the four corners of the field, or to either side of the midfield line. Notice that in this case, the agents consider passing to *locations* on the field rather than to actual players. Nonetheless, the trained pass-evaluation DT can be used *as if* there were a teammate in the proposed field location. The action space is illustrated in Figure 7.2.

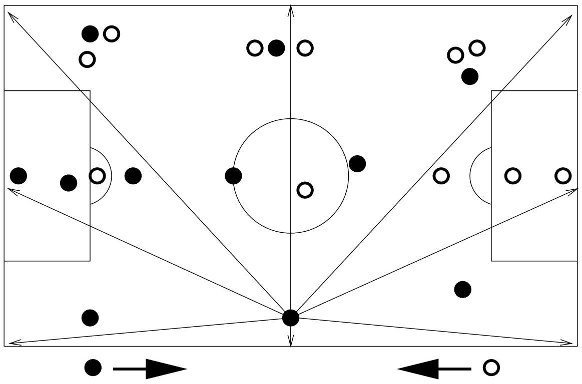

Figure 7.2: The action space used in this chapter. The black and white dots represent the players attacking the right and left goals respectively. Arrows indicate a single player's (the player from which the arrows emanate) action options when in possession of the ball. The player kicks the ball towards a fixed set of markers around the field, including the corner markers and the goals.

We extend the definition of Φ first presented in Section 6.1.1 to cover the action space used in this section as follows. Assume that for the action $a \in A$ (a kick to a location on the field), the DT predicts class κ with confidence $\gamma \in [0, 1]$ when the agent is in state s. The DT is evaluated as if there were a teammate at the kick destination corresponding to action

a. Then[2]

$$\Phi(s,a) = \begin{cases} \gamma & \text{if } \kappa = \text{S (success)} \\ 0 & \text{if } \kappa = \text{M (miss)} \\ -\gamma & \text{if } \kappa = \text{F (failure)} \end{cases}$$

The two previously-learned behaviors (see Chapters 5 and 6) are both trained off-line in limited, controlled training situations. They can be trained in such a manner due to the fact that they only involve a few agents: ball interception only depends on the ball's and the agent's motions; passing only involves the passer, the receiver, and the agents in the immediate vicinity. On the other hand, deciding where to pass the ball during the course of a game requires training in real games since the value of a particular action can only be judged in terms of how well it works when playing with particular teammates against particular opponents. For example, as presented in Section 7.1.1, passing backwards to a defender could be the right thing to do if the defender has a good action policy, but the wrong thing to do if the defender is likely to lose the ball to an opponent.

Although our trained DT accurately predicts whether a player can execute a pass, it gives no indication of the strategic value of doing so. But the DT reduces a detailed state description to a single continuous output. It can then be used to drastically reduce the complex state and provide a useful state generalization. In this work we use the DT to define the crucial action-dependent feature function e in TPOT-RL. Thus, in the context of our layered learning implementation, the new learned layer (pass selection) uses the previous layer (pass evaluation) as part of its input representation.

7.3.1 State Generalization Using a Learned Feature

As is the case throughout this thesis and as illustrated in Figure 7.2, the team formation is divided into 11 positions. Thus,

$$\begin{aligned} M &= \text{the team's set of positions (roles) } (|M| = 11) \\ P(s) &= \text{the player's own current position} \end{aligned}$$

Using the layered learning approach, we use the previously trained DT to define e, which is the main component of the input representation used to learn Q. In particular, we use Φ to cluster actions into two sets indicating predicted success or failure:

$$\begin{aligned} U &= \{Success, Failure\} \\ e_\Phi(s,a) &= \begin{cases} Success & \text{if } \Phi(s,a) \geq C \\ Failure & \text{if } \Phi(s,a) < C \end{cases} \end{aligned}$$

In our experiments, we use $C = .734$ as the threshold for clustering action. We find that this threshold clusters the predictions into classes of roughly equal size.

[2]In Chapter 6, the parameter s was implicit for consistency of notation within the chapter. Similarly, for notational consistency within this chapter, the fact that Φ is evaluated from the perspective of the "passer"—the agent that is acting—is implicit.

According to these definitions, $|U| = 2$ and $V = U^8 \times \{PlayerPositions\}$ so $|V| = |U|^{|A|} * |M| = 2^8 * 11$. Under the assumption that $Q(s, a_i)$ depends only on $e(s, a_i)$, the total number of independent Q-values is $|U| * |M| * |A| = 2 * 8 * 11$.

This feature space is immensely smaller than the original state space, which has more than 10^{198} states (see Section 2.2). Since e indicates the likely success or failure of each possible action, at action-selection time, we only consider the actions that are likely to succeed:

$$W = \{Success\}$$

. Therefore, each agent learns $|W|*|A| = 8$ Q-values, with a total of 88 ($|W|*|A|*|M|$) learned by the team as a whole. Even though each agent only gets about 10 training examples per 10-minute game and the reward function shifts as teammate policies improve, such a learning task is tractable.

In addition to e_Φ, we define two other action-dependent feature functions for the purpose of comparison:

- e_r is a random function, returning $Success$ or $Failure$ randomly.
- e_h is a hand-coded heuristic pass-evaluation function based on one defined in Appendix B (Section B.2.5) that we successfully used on our real robot team.

7.3.2 Value Function Learning via Intermediate Reinforcement

As in any RL approach, the reward function plays a large role in determining what policy is learned when using TPOT-RL. One possible reward function, R_g is based entirely upon reaching the ultimate goal. In the soccer domain, we define R_g in terms of goals scored. If a goal is scored t time steps after action a is taken ($t \leq t_{lim}$), then the reward is $\pm Q_{max}/t$ depending on whether the goal is scored for or against. In this way, an agent charts the actual (long-term) results of its policy in the environment. Notice that the reward is discounted based on how long after acting the rewarding state is achieved.

Although goals scored are the true rewards in this domain, such events are very sparse. In order to increase the feedback from actions taken, it is useful to use an intermediate reinforcement function, which provides feedback based on intermediate states towards the goal [Mataric 94a]. Without exploring the space of possible such functions, we created one reward function R_i using intermediate reinforcement.

Like R_g, R_i gives rewards for goals scored. However, agents also receive rewards if the ball goes out of bounds, or else after a fixed period of time t_{lim} based on the ball's average lateral position on the field. In particular, when an agent takes action a_i in state s such that $e(s, a_i) = u$, the agent notices the time t at which the action was taken as well as the x-coordinate of the ball's position at time t, x_t. The reward function R_i (like R_g) takes as input the observed ball position over time t_{lim} (a subset of $S^{t_{lim}}$) and outputs a reward r. Since the ball position over time depends on other agents' actions, the reward is stochastic and non-stationary. Under the following conditions, the agent fixes the reward r:

1. if the ball goes out of bounds (including a goal) at time $t + t_o$ ($t_o < t_{lim}$);
2. if the ball returns to the agent at time $t + t_r$ ($t_r < t_{lim}$);

3. if the ball is still in bounds at time $t + t_{lim}$.

In case 1, the reward r is based on the value r_o as indicated in Figure 7.3:

$$r = \frac{r_o}{1 + (\phi - 1) * t_o/t_{lim}}. \tag{7.3}$$

Thus, the farther in the future the ball goes out of bounds (i.e. the larger t_o), the smaller the absolute value of r. This scaling by time replaces the discount factor used in Q-learning. We use:

$$\begin{aligned} t_{lim} &= 30 \text{ seconds (300 simulator cycles)} \\ Q_{max} &= 100 \\ \phi &= 10 \end{aligned}$$

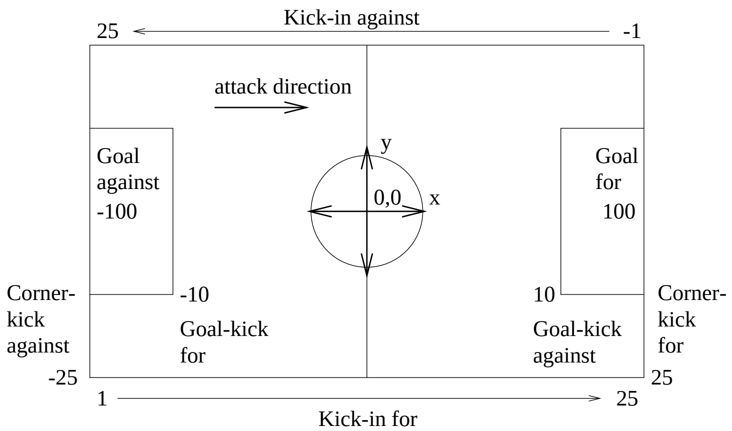

Figure 7.3: The component r_o of the reward function R_i based on the circumstances under which the ball goes out of bounds. For kick-ins, the reward varies linearly with the x-position of the ball.

In cases 2 and 3, the reward r is based on the average x-position of the ball over the time t to the time $t+t_r$ or $t+t_{lim}$. Over that entire time span, the agent samples the x-coordinate of the ball at fixed, periodic intervals and computes the average x_{avg} over the times at which the ball position is known. Then if x_{og} is the x-coordinate of the opponent goal (the right goal in Figure 7.3) and x_{lg} is the x-coordinate of the learner's goal:

$$r = \begin{cases} \phi * \frac{x_{avg} - x_t}{x_{og} - x_t} & \text{if } x_{avg} > x_t \\ -\phi * \frac{x_t - x_{avg}}{x_t - x_{lg}} & \text{if } x_{avg} \leq x_t \end{cases} \tag{7.4}$$

Thus, the reward is the fraction of the available field by which the ball is advanced, on average, over the time-period in question. Note that a backwards pass can lead to positive reward if the ball then moves forward in the near future and conversely, a forwards pass can lead to a negative reward. The use of parameter ϕ in both Equations 7.3 and 7.4 insures that intermediate rewards cannot override rewards for attaining the ultimate goal, Q_{max} which is the maximum value of r_o in Equation 7.3.

When using either R_g or R_i, the reward r is based on direct environmental feedback. R_g passes discounted reward back to the agents only when the world enters a state with which some reward is associated. R_i is a domain-dependent intermediate reinforcement function based upon heuristic knowledge of progress towards the goal.

Notice that R_i relies solely upon the agent's own impression of the environment. If it fails to notice the ball's position for a period of time, the intermediate reward is affected (the agents always notice when the ball goes out of bounds or into a goal via an aural message from the referee). However, agents can track the ball much more easily than they can deduce the internal states of other agents as they would have to do were they to determine future team state transitions.

Finally, after taking action a_i and receiving reward r, $Q(e(s, a_i), a_i)$ is updated according to equation 7.2 with the learning rate

$$\alpha = .02$$

Thus, even though we average all reward values achieved as a result of taking an action in a given state, each new example accounts for 2% of the updated Q-value: rewards achieved further in the past are weighted less heavily.

7.3.3 Action Selection for Multi-Agent Training

One characteristic of some multi-agent domains that makes them non-stationary is the fact that multiple agents are concurrently learning. Thus, from each individual's perspective, the environment is not a stable system. In order to deal with this challenge, we adopt two action strategies:

- Each agent stays in the same state partition throughout training;
- Exploration rate is very high at first and gradually decreases simultaneously for all agents;

By having each agent remain in the same partition throughout training, we are, in effect, distributing training into $|M|$ partitions, each with a lookup-table of size $|A| * |U|$. After training, each agent can be given the trained policy for all of the values of M, enabling the agents to move through the entire state space. Were each agent required to learn the policies of all positions, training would take at least $|M|$ times longer[3].

As in all RL paradigms, the tradeoff between exploration and exploitation is potentially problematic. Especially since rewards in our case are stochastic and feature values encode large numbers of states, early exploitation runs the risk of ignoring the best possible actions in certain states. As a result, when in state s, our agents choose the action with the highest Q-value—action a_i such that $\forall j$, $Q(f(s), a_i) \geq Q(f(s), a_j)$—with probability p, and a random action with probability $1 - p$. In all of our training runs, p gradually increases from 0 to .99.

[3]Training would likely take even longer than that. Since teammates would all be learning more slowly as well, it would take longer for the agents to receive informative reinforcement values.

7.3.4 Results

Empirical testing demonstrates that TPOT-RL can effectively learn multi-agent control policies with few training instances in a complex, dynamic domain. Unless otherwise noted, for all experiments reported in this section:

- The learning agents start out acting randomly and with empty Q-tables: $\forall v \in V, a \in A, Q(v, a) = 0$.
- Over the course of learning, the probability of acting randomly, p, decreases linearly over periods of 40 games from 1 to .5 in game 40, to .1 in game 80, to .01 in game 120 and thereafter.
- The learning agents use the intermediate reward function R_i.

Figure 7.4 plots cumulative goals scored by a learning simulated soccer team playing against an otherwise equally-skilled team that passes to random destinations over the course of a single long run equivalent in time to 160 10-minute games. As apparent from the graph, the team using TPOT-RL learns to vastly outperform the randomly passing team. Other than the pass decisions, the behaviors of the agents on the randomly passing team are identical to those on the learning team. During this experiment, $|U| = 1$, thus rendering the function e irrelevant: the only relevant state feature is the player's own position on the field.

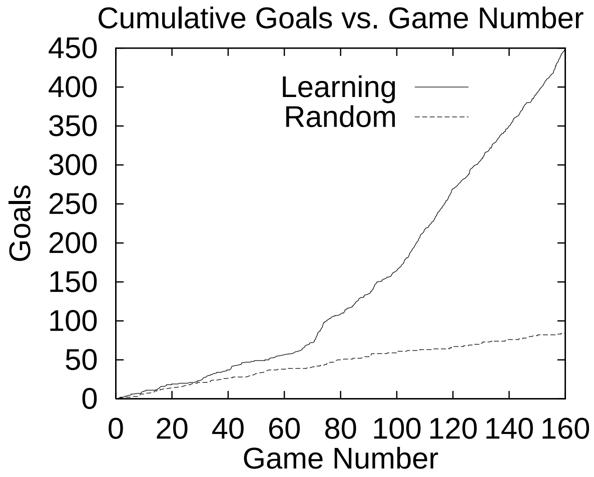

Figure 7.4: Total goals scored by a learning team playing against a randomly passing team. The independent variable is the number of 10-minute games that have elapsed.

A key characteristic of TPOT-RL is the ability to learn with minimal training examples. During the run graphed in Figure 7.4, the 11 players get an average of 1490 action-reinforcement pairs over 160 games. Thus, players only get reinforcement an average of 9.3 times each game, or less than once every minute. Since each player has 8 actions from which to choose, each action is only tried an average of 186.3 times over 160 games, or just over once every 10-minute game. Under these training circumstances, very efficient learning is clearly needed.

TPOT-RL is effective not only against random teams, but also against goal-directed, hand-coded teams. For testing purposes, we construct an opponent team which plays with all of its players on the same side of the field, leaving the other side open as illustrated by the white team in Figure 7.2. The agents use a hand-coded RCF which directs them to pass the ball up the side of the field to the forwards who then shoot on goal. The team switches from one side of the field to the other every 5 minutes of simulation (half of a game). We call this team the *switching team.* Note that the switching team is similar to a team that switches between the OPR and OPL (only play right/left) strategies defined in Chapter 6. The only difference is that the switching team uses a goal-directed, rather than a random, RCF.

Were the opponent team to always stay on the same side of the field, as is the case with the goal-directed OPR team, the learning team could advance the ball up the other side of the field without any regard for current player locations. Thus, TPOT-RL could be run with $|U| = 1$, which renders e inconsequential. Indeed, we verified empirically that TPOT-RL is able to learn an effective policy against the OPR team using $|U| = 1$. As shown in Figure 7.5(b), after 160 games of learning, the learning team is able to beat the OPR team by a cumulative score of 259–124 over the course of 40 test games. As a point of comparison, Figure 7.5(a) shows the result of always shooting towards the opponent goal—a reasonable heuristic—when playing against the OPR team.

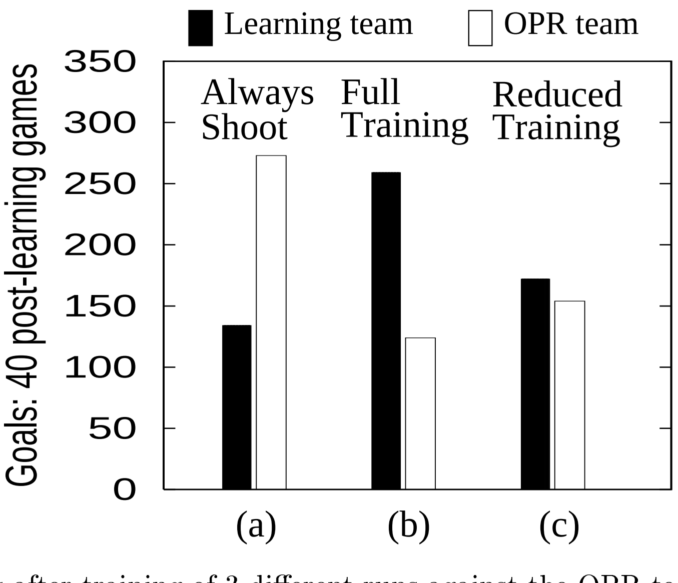

Figure 7.5: The results after training of 3 different runs against the OPR team (always shown as the white bar). (a) shows the result of always executing the shoot action. (b) and (c) show the results when training for 160 and 80 games respectively when using TPOT-RL with $|U| = 1$.

Also against the OPR team, we tested the effect of training time on the effectiveness of the learned policy. The results can be seen by comparing Figures 7.5(b) and 7.5(c). These two trials are identical except that the latter used an accelerated training schedule: The exploration rate was decreased over 20-game intervals for a total of 80 games rather than 40-game intervals for a total of 160 games. The reduced training leads to worse performance over 40 subsequent test games. The purpose of this experiment was to verify the need for on the order of 160 training games to adequately evaluate TPOT-RL's performance. Further testing indicated that training beyond 160 games does not significantly affect performance. All other trials reported below use 160 games of training.

Exploiting the Previously Learned Features

Against the switching team, an agent's best action depends on the current state: in general, the agent should pass away from the side of the field on which the opponents are located. Thus an action-dependent feature function that discriminates among possible actions dynamically can help TPOT-RL. Figure 7.6 compares TPOT-RL with different functions e and different sets W when learning against the switching team.

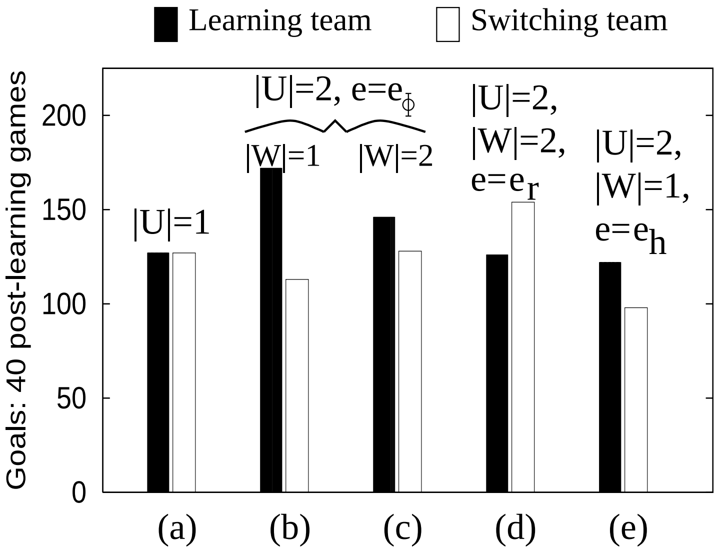

Figure 7.6: The results after training of 5 different TPOT-RL runs against the switching team.

With $|U| = 1$ (Figure 7.6(a)), the learning team is unable to discriminate among states in which passes are likely to succeed or fail since each agent has only one Q-value associated with each possible action[4]. With $|U| = 1$, it loses 139–127 (cumulative score over 40 games after 160 games of training).

In contrast, with the previously trained DT classifying passes as likely successes or failures (e_Φ) and with filtering out the failures ($W = \{Success\}$), the learning team wins 172–113 (Figure 7.6(b)). Therefore the learned pass-evaluation feature is able to usefully distinguish among possible actions and help TPOT-RL to learn a successful action policy. The DT also helps learning when $W = U$ (Figure 7.6(c)), but when $W = \{Success\}$ performance is better.

Figure 7.6(d) demonstrates the value of using an informative action-dependent feature function e. When a random function e_r is used, TPOT-RL performs noticeably worse than when using e_Φ. For e_r we show $|W| = 2$ because it only makes sense to filter out actions when e contains useful information. Indeed, when $e = e_r$ and $|W| = 1$, the learning team performs even worse than when $|W| = 2$ (it loses 167–60). The DT even helps TPOT-RL

[4]Notice that whenever $|U| = 1$, it must be the case that $W = U$.

more than a hand-coded heuristic pass-evaluation function (e_h) based on the one defined in Appendix B (Section B.2.5) that we successfully used on our real robot team (Figure 7.6(e)).

Final score is the ultimate performance measure. However, we examined learning more closely in the best case experiment ($e = e_\Phi$, $W = \{Success\}$—Figure 7.6(b)). Recall that the learned feature provides no information about which actions are *strategically* good. TPOT-RL must learn that on its own. To test that it is indeed learning to advance the ball towards the opponent's goal (other than by final score), we calculated the number of times each action is predicted to succeed ($e_\Phi(s, a) = Success$) and the number of times it was actually selected by TPOT-RL after training. Throughout the entire team, the 3 of 8 (37.5%) actions towards the opponent's goal were selected 6437/9967 = 64.6% of the times that they were predicted to succeed. Thus TPOT-RL learns that it is, in general, better to advance the ball towards the opponent's goal.

Verifying that the filter is eliminating action choices based on likelihood of failure, we found that 39.6% of action options were filtered out when $e = e_\Phi$ and $|W| = 1$. Out of 10,400 action opportunities, it was never the case that all 8 actions were filtered out: in all cases, $B(s) \neq \emptyset$.

Comparing Reward Functions

The reward function R_i, used in all the experiments reported above, is engineered to increase the amount of reinforcement information available to the agents as opposed to the extremely sparse reinforcements available from goals scored (R_g). Nevertheless, while not as effective as R_i, R_g does provide enough reinforcement for agents to learn effective policies. Using the same 160-game training paradigm and using $|U| = 1$, a team learning with R_g is able to beat the OPR team (Figure 7.7(a)), though by considerably less than the team using R_i in otherwise identical circumstances (Figure 7.7(b)—the same trial as shown in Figure 7.5(b)). Therefore, the intermediate reinforcement reward function, R_i is a key component to achieving successful results with TPOT-RL.

7.4 TPOT-RL Applied to Network Routing

Section 7.3 demonstrates that TPOT-RL can be used to learn effective behaviors in one team-partitioned, opaque-transition domain. However, in creating TPOT-RL, we intend it as a new general multi-agent learning algorithm. This section demonstrates that it can indeed apply beyond simulated robotic soccer.

We identify network routing as another team-partitioned, opaque transition domain. As presented in Section 2.4, in our formulation of network routing, each network node is considered as a separate agent. The agents act as a team as they try to cooperate in sending packets through the network as efficiently as possible. The domain is team-partitioned since each agent learns only a policy at its own node: the function P partitions the state space based, in part, on the node at which each agent is situated. Prior research indicates that distributed network control is advantageous even in high-speed networks (such as ATM networks) in which centralized control is possible [Horikawa *et al.* 96].

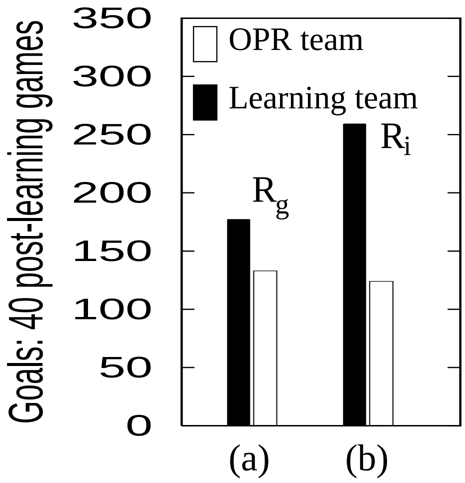

Figure 7.7: The results after training of 2 different runs against the OPR team. (a) shows the result of learning with the R_g reward function. (b) shows the result of training with the R_i reward function. In both cases, $|U| = 1$.

Network routing is opaque-transition because agents cannot see a packet's route after sending it along a link. Like in robotic soccer, agents' actions are chained, with each agent able to affect which agent will act *next*, but with no control beyond that. Agents get no short-term reward for their actions and cannot track the transitions in the environment.

Nonetheless, the team of agents can learn to effectively route packets by observing local state information (network traffic). The action-dependent feature function e in our implementation of TPOT-RL for network routing provides useful local information that correlates with the long-term reward: e returns the amount of recent network traffic on the links leading from an agent's node. In our experiments, we assume that the agents get intermittent long-term performance statistics transmitted back to them. Thus they are able to learn to collaborate.

Aside from being team-partitioned and opaque-transition, network routing is also similar to robotic soccer in that the world changes dynamically in a manner beyond the team's control. In robotic soccer, opponents can change their strategies; in network routing the distribution of packets introduced into the network can change.

In this section, I present in detail our application of TPOT-RL to network routing, a second multi-agent learning task. In Sections 7.4.1–7.4.3, I provide the algorithmic details of our implementation of TPOT-RL. Section 7.4.4 specifies our experimental methodology including the parameters used in the network routing simulator and Section 7.4.5 presents detailed empirical results demonstrating the effectiveness of TPOT-RL in the network routing domain.

7.4.1 State Generalization

As defined in Section 7.2.1, the function $f : S \mapsto V$ generalizes the state space based on two components: an action-dependent feature function $e : S \times A \mapsto U$ and a coarse partitioning function $P : S \mapsto M$. Given a state $s \in S$ from which the agent at node n_i is faced with the decision of routing packet k_j:

$$\begin{aligned} M &= N \times N \quad (|M| = m^2) \\ P(s) &= (n_i, k_{j_{dest}}) \end{aligned}$$

Recall that, as defined in Section 2.4, m is the number of nodes in the network: $|N| = m$. Using this partitioning function P, the agent at node n_i learns to act only in the cases that $P(s) = (n_i, k_{j_{dest}})$ for some j: the space is partitioned evenly among the m agents, with each getting m partitions.

The action-dependent feature function $e : S \times A \mapsto U$ is defined as follows. A is the set of actions available and is represented in terms of the nodes to which a packet can be sent. Thus $A = N$. However, the agent at node n_i may only use the actions in $L_{n_i} \subseteq N$—the set of links from node n_i. The elements of

$$U = \{high, low\}$$

reflect the network activity over a particular link in the last `activity_window` time units. An agent can store the link activity along all of the links from its node since it is either the sender or the recipient of all packets sent along these links.

Define $\tau(s, a, \texttt{activity\_window})$ as the number of packets sent along the link corresponding to action a in the last `activity_window` time units divided by `activity_window`. Then if action a is the act of sending a packet along link l,

$$e(s,a) = \begin{cases} high & \text{if } \tau(s, a, \texttt{activity\_window}) \geq C \\ low & \text{if } \tau(s, a, \texttt{activity\_window}) < C \end{cases}$$

We use $\texttt{activity\_window} = 100$ and $C = .5$. Notice that e is an action-dependent function since it depends on the proposed action of sending a packet along link l. It is also based entirely upon local information available to agents that maintain internal state, collecting traffic statistics over time.

Like in the robotic soccer example, this state generalization drastically reduces a huge state space to the point that agents can store Q-values in a lookup table. In our implementation, according to Equation 2.2, the entire state space in our experiments has more than 10^{3000} states ($m = 12$,$C_{net} = 1000$). However, with $|U| = 2$, $|A| \leq 3$, and $|M| = m^2$ the total number of Q-values for the team to learn is at most $6m^2$, with each agent learning no more than $6m$ values. In our experiments reported below, $m = 12$. Therefore each agent must learn only 72 Q-values.

7.4.2 Value Function Learning

As per equation 7.2, agents learn $Q(v, a)$—the value of taking action a when in a state s such that $f(s) = v$—by receiving a reward r via a reward function R. In this case, if v indicates that node n is trying to send a packet k on its way to node k_{dest} ($P(s) = (n, k_{dest})$), then $Q(v, a)$ is meant to estimate the time that it will take for the packet to reach node k_{dest}. Thus agents aim to take actions that will lead to *minimal* r.

Unlike the soccer domain, in which agents can observe the ball's progress over time, network routing does not provide much opportunity for intermediate reinforcement. R is almost entirely based on the actual time that the packet k takes to travel from node n_i to its destination k_{dest}.

When k successfully arrives at k_{dest}, the agent at node k_{dest} can examine the times at which it left each node along its path from k_{source} as stored in k_{path}. From this information, it can deduce the time taken from each node along the path given the action taken at that node. Then periodically, every `update_interval` seconds, the nodes in the network update each other on the long-term results of their actions.

Thus, after the agent at node n_i sends a packet k_i along link l at time $t_{k_i,l}$, the agent at node n_i receives reward r equal to the time it took for the packet to eventually reach its destination $k_{i_{dest}}$:

$$r = k_{i_{dtime}} - t_{k_i,l}$$

In this case, the goal of each node is to minimize its reward r. Notice that this formulation of the reward function R is entirely goal oriented—in general, there is no opportunity for the agent at node n_i to observe a packet's progress on the way to its destination.

However, there *is* one exception. Especially at the early stages of learning when actions are mostly random, a packet often returns to a node from whence it came at some interval t later than it last left the node. In this situation, the agent at the node in question infers that the previous action a taken on this packet was ineffective and generates an intermediate reward signal $r = Q(v, a) + t$, thus increasing the cost estimate $Q(v, a)$.

To put a bound on the timing of rewards, the reward r is bounded by Q_{max}. That is, if a node takes longer than Q_{max} to arrive, $r = Q_{max}$. A node can assume that the packet did not arrive in this time if it has not heard about its arrival after $Q_{max}+$ `update_interval` simulated seconds. Therefore, $t_{lim} = Q_{max}+$ `update_interval`. In our experiments, we use

$$\begin{aligned} Q_{max} &= 2000 \\ \texttt{update\_interval} &= 10 \\ t_{lim} &= 210 \end{aligned}$$

Finally, just as in the robotic soccer implementation of TPOT-RL, after the agent at node n takes action a on a packet k destined for node k_{dest} and receives reward r, $Q((e(s, a), (n, k_{dest})), a)$ is updated according to equation 7.2 with the learning rate

$$\alpha = .02$$

Thus, again, even though we average all reward values achieved as a result of taking an action in a given state, each new example accounts for 2% of the updated Q-value: rewards achieved further in the past are weighted less heavily.

7.4.3 Action Selection

In our network routing implementation of TPOT-RL, Q-values are all initialized to low values (0 or the shortest path length between nodes) before learning and agents always choose the action with the lowest Q-value. Thus, each action is tried at least once, but there is no deliberate exploration. We find this total exploitation strategy to be effective, presumably because as unsuccessful actions are repeated, their costs increase due to network congestion, thus naturally causing agents to try the other alternatives periodically. However, should exploration become necessary, we could easily switch to a probabilistic action-selection strategy as we use in the robotic soccer implementation.

We do not experiment with action filtering in this domain, finding that it is not necessary to achieve good results. Thus, in all of our experiments,

$$W = U$$

7.4.4 Experimental Setup

In this section I lay out the details of our experimental setup for testing the effectiveness of TPOT-RL in the network routing domain. The detailed empirical results are then reported in Section 7.4.5.

All of the experiments use a network architecture (node and link patterns) as shown in Figure 7.8. The nodes are numbered for reference in the text. Packets are injected into the network at random intervals according to a Poisson distribution at an average rate of c per simulated second. In our experiments, we use $c = 3$. We create three different traffic patterns within this network by controlling the distributions of sources and destinations of injected packets. Our traffic patterns are controlled by two variables: p_6, the probability that a new packet is destined for node 6 (see Figure 7.8); and f_s, the frequency with which the traffic patterns switches (i.e. number of simulated seconds between pattern switches). In our experiments, we use $p_6 = .25$ and $f_s = 10{,}000$. The three traffic patters we define are:

Top-heavy: With probability p_6, the injected packet has node 6 as its destination and a random source; with probability $1 - p_6$, the injected packet has source and destination chosen randomly (without replacement) from the set of nodes $\{1,2,3,4,5\}$.

Bottom-heavy: With probability p_6, the injected packet has node 6 as its destination and a random source; with probability $1-p_6$, the injected packet has source and destination chosen randomly from the set of nodes $\{7,8,9,10,11\}$.

Switching: Every f_s simulated seconds, the traffic pattern switches between the top-heavy and bottom-heavy patterns.

As laid out in Section 2.4, there are several parameters governing the timing and capacities of network traffic. We use $C_{node} = C_{net} = 1000$, $t_n = t_l = 1.0$.

In our experiments, we test several different packet routing strategies under the different traffic patterns defined above. I define the strategies in terms of what the agent at node n does when trying to route packet k to its destination k_{dest}. It must choose from among the possible links in L_n.

Random (RAND): k is sent along a random link $l \in L_n$.

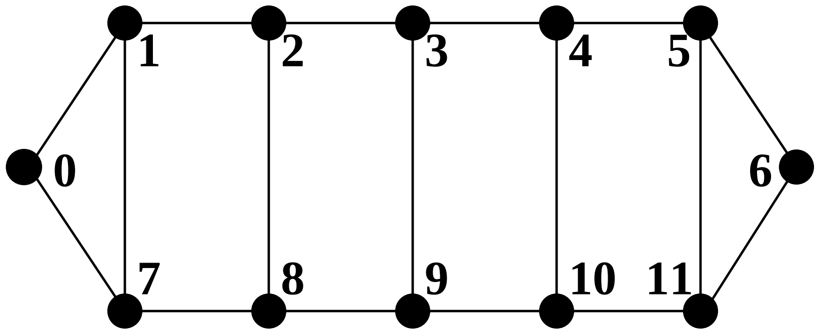

Figure 7.8: The network architecture used for our experiments. The nodes are numbered for reference in the text.

Shortest (SHRT): k is sent along the link that would get it to k_{dest} in the fewest number of hops. Shortest paths are precomputed and stored based on the network topology. If more than one link would lead along paths of the same shortest length, one such link is chosen randomly.

Hand-coded (HAND): We created a policy by hand designed to work well with the top-heavy traffic pattern.

- When $k_{dest} = 6$, if $n \in \{7, 8, 9, 10, 11\}$, k is sent along the shortest path towards node 6 (along the bottom of the network in Figure 7.8). Otherwise, ($n \in \{0, 1, 2, 3, 4, 5\}$) k is sent to a node in the set $\{6, 7, 8, 9, 10, 11\}$ from where it can then continue along the shortest path to node 6. Note that in all cases k can be sent to a node in this set in one hop.
- When $k_{dest} \in \{1, 2, 3, 4, 5\}$, k is sent along the shortest path (along the top of the network).
- Similarly, when $k_{dest} \in \{7, 8, 9, 10, 11\}$, k is sent along the shortest path (along the bottom of the network).

We expect this policy to do fairly well with the top-heavy traffic pattern since the traffic is distributed fairly evenly among the top and bottom portions of the network. Packets headed for node 6 ($p_6 = 25\%$ of the packets) use the bottom, while other packets use the top of the network.

Q-routing (QROUT): This strategy is the Q-routing algorithm introduced in [Littman and Boyan 93]. We use exactly the same implementation for testing. There are two variants. One that starts with all Q-values initialized to 0, and another that initializes all Q-values to be the shortest distance between nodes. In the former case, the policy at the beginning is equivalent to RAND; in the latter, it is equivalent to SHRT. In both cases, the Q-values can change freely during learning. We use the default learning rate of $\eta = 0.7$.

TPOT-RL: This strategy is the one described in detail in Sections 7.4.1–7.4.3. Like QROUT, TPOT-RL can be initialized with Q-values at 0 or with the length of the shortest path between nodes. In the latter case, the initialization is immediately overwritten by the first real reward signal. However, the initialization guides the initial agent decisions.

In addition, stored Q-routing and TPOT-RL policies can be loaded and used exclusively with no further learning allowed. We use this technique for testing purposes.

A major difference between Q-routing and TPOT-RL is that when using Q-routing, neighboring nodes send back their own value estimates after every packet transfer. Thus Q-routing generates a considerable amount of additional traffic. On the other hand, we assume that agents do not know anything about the network beyond their own nodes and links. Performance statistics are propagated through the network in batches periodically, allowing for a tradeoff between overhead packets and learning rate. However, unlike Q-routing agents, TPOT-RL agents are unable to use dynamic programming.

7.4.5 Results

This section presents detailed empirical results verifying the effectiveness of TPOT-RL in the network routing domain. First I present comparisons of the different routing strategies in a top-heavy traffic pattern. Then I describe the effects of testing the resulting policies on the bottom-heavy traffic pattern. Finally, I present our results from training policies on the switching traffic and testing their generalization across the three traffic patterns.

Top-Heavy Traffic

First, we tested the five different routing strategies under top-heavy network traffic conditions. I chart both the average delivery time of the packets and the average number of hops per packet. Results are tabulated over intervals of 100 simulated seconds. Because of the difference in scales of the performance of, on the one hand RAND and SHRT, and on the other hand HAND, QROUT, and TPOT-RL, I break the results into two sets of graphs. Figures 7.9 and 7.10 compare the results of using the RAND, SHRT, and TPOT-RL routing strategies in terms of average delivery time and average number of hops respectively. Notice that as the network fills up, the RAND performs worse and worse in both respects. By definition, SHRT produces the minimum possible number of hops per packet, but again exhibits a continual increase in average delivery time as the nodes' packet queues lengthen.

In this and all other TPOT-RL runs, unless specified otherwise, `update_interval = 10`, $Q_{max} = 2000$, and $|U| = 1$: like in robotic soccer, under constant conditions, TPOT-RL can be effective even without the help of action-dependent features.

Figures 7.11 and 7.12 are drawn on a much smaller y-axis scale, as the HAND, QROUT, and TPOT-RL strategies all vastly outperform the other two (at least in terms of average delivery time). In these runs, QROUT and TPOT-RL are both initialized with the shortest paths to eliminate the big spike that would otherwise throw off the scale at the beginning of the graphs as the algorithms begin acting randomly. When these algorithms start out with Q-values initialized to 0, the right hand parts of the curves look qualitatively the same.

Again, the average number of hops does not correlate with the average delivery time, the real measure of interest. TPOT-RL outperforms both HAND and QROUT by sending some

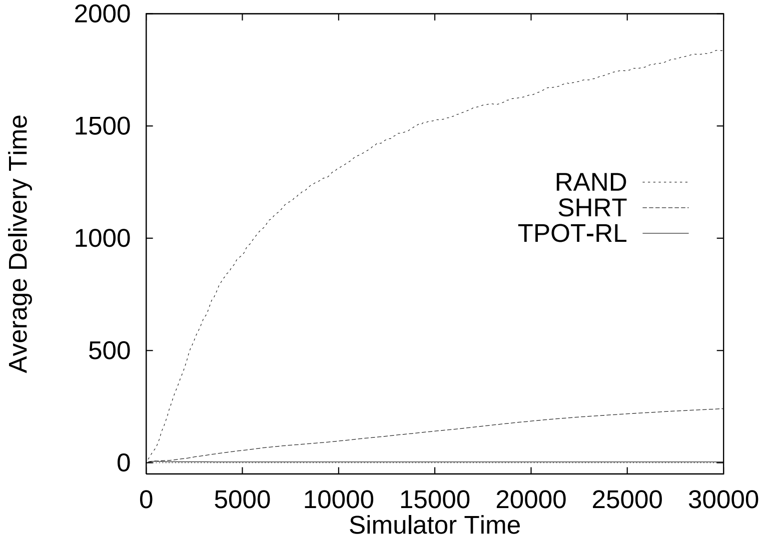

Figure 7.9: Average delivery time of packets in a network with the top-heavy traffic pattern when using three different routing strategies: RAND, SHRT, and TPOT-RL.

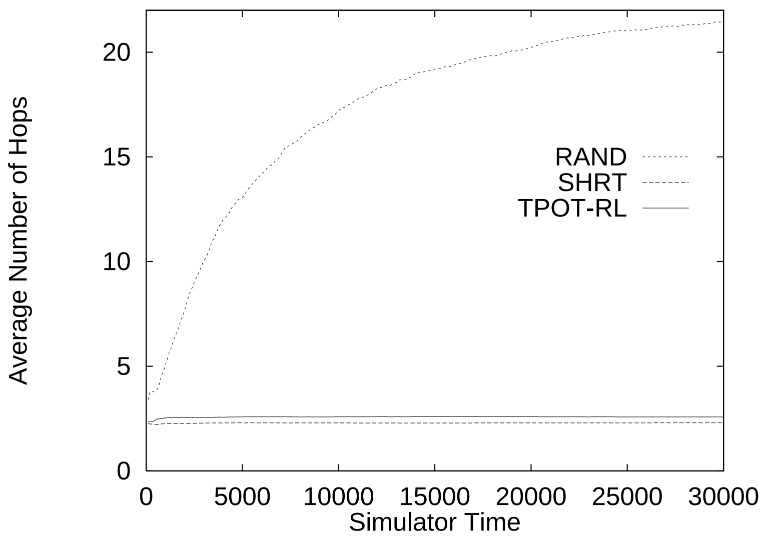

Figure 7.10: Average number of hops for packets in a network with the top-heavy traffic pattern when using three different routing strategies: RAND, SHRT, and TPOT-RL.

packets along longer, but less congested paths. The remainder of the results in this section are presented only in terms of average packet delivery time.

Bottom-Heavy Traffic

Figure 7.13 illustrates the results of running policies designed for top-heavy traffic under bottom-heavy conditions. The solid bars show average traffic delivery time for SHRT, HAND, QROUT, and TPOT-RL under top-heavy traffic. These numbers are the same as the end-results shown in Figures 7.9 and 7.11. As noticed above, the HAND, QROUT, and TPOT-RL strategies all perform well, having been designed or trained for these conditions.

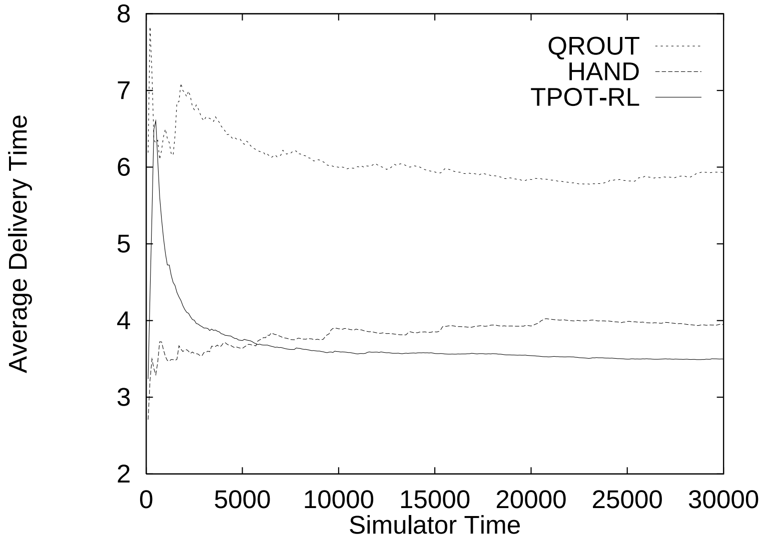

Figure 7.11: Average delivery time of packets in a network with the top-heavy traffic pattern when using three different routing strategies: QROUT, HAND, and TPOT-RL.

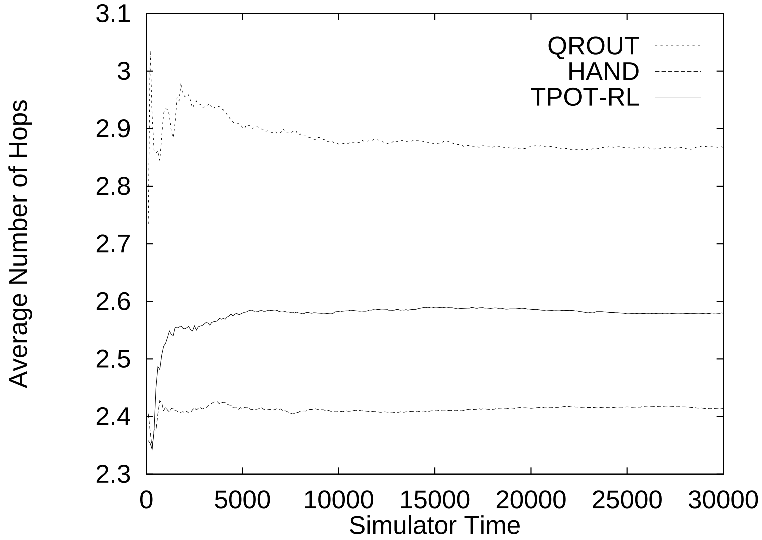

Figure 7.12: Average number of hops for packets in a network with the top-heavy traffic pattern when using three different routing strategies: QROUT, HAND, and TPOT-RL.

However, when the resulting policies are tested under bottom-heavy traffic conditions, none of them perform well. In these runs, shown with hollow bars, the policies from the ends of the previous runs are used with no additional learning. Of course the SHRT and HAND policies, which do not learn, are constant throughout both runs.

I use a logarithmic scale on the y axis in Figure 7.13 in order to accommodate the large discrepancy in values while still illustrating the differences.

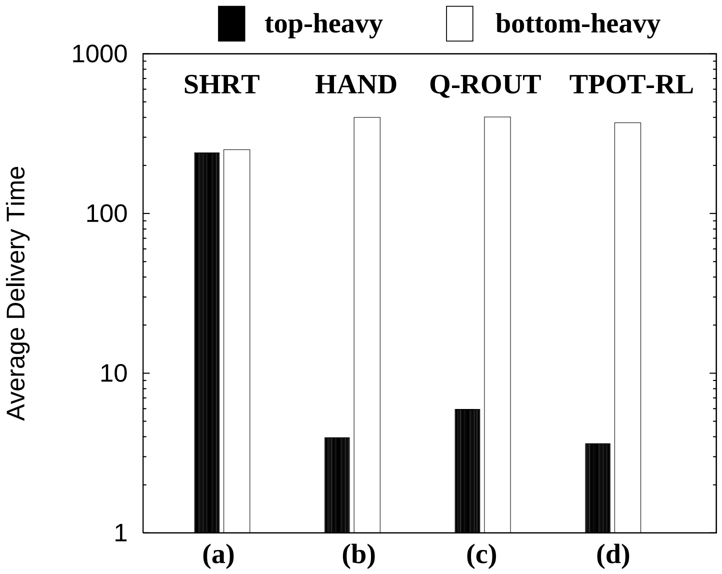

Figure 7.13: Fixed policies running in the top-heavy and bottom-heavy traffic patterns. The QROUT and TPOT-RL policies are both trained under top-heavy conditions. Note the log scale on the y axis.

Switching Traffic

While we would not expect a policy trained exclusively under top-heavy network traffic conditions to generalize to bottom-heavy conditions, we would like a policy trained under switching conditions, since it includes periods of both of the other traffic patterns, to generalize across all three. Figure 7.14 shows the results of HAND, QROUT, and two variants of TPOT-RL under all three traffic patterns. In all of the learning cases, the policy is first trained under switching network traffic for 30,000 simulated seconds and then fixed for testing. In all cases, the solid bar shows results for top-heavy traffic, the hollow bar for bottom-heavy traffic, and the striped bar for switching traffic. Again, I use a logarithmic scale on the y axis for presentation purposes.

The two TPOT-RL runs differ only in the set U used. When $|U| = 1$, no difference is made based on local traffic conditions: routing decisions are based entirely upon the packet's destination. When $|U| = 2$, agents can implement different policies based on different local traffic conditions. Note that Q-routing always routes packets based solely on their destinations.

Figure 7.14 clearly shows the advantage of TPOT-RL. Since HAND is designed explicitly for top-heavy traffic, it fails as expected in other conditions (Figure 7.14(a)). Similarly, QROUT is forced to adapt to one traffic pattern at the expense of the other (Figure 7.14(b)). In this case, it clearly optimizes its performance under bottom-heavy conditions, sacrificing its performance under top-heavy conditions, and thus under switching conditions as well.

On the other hand, TPOT-RL with $|U| = 1$ (Figure 7.14(c)) finds a middle ground in which the network performs fairly well under all traffic conditions, but never as well as is possible. It is when $|U| = 2$ (Figure 7.14(d)) that TPOT-RL can take advantage of its action-dependent feature function to find correlations between local information and long-

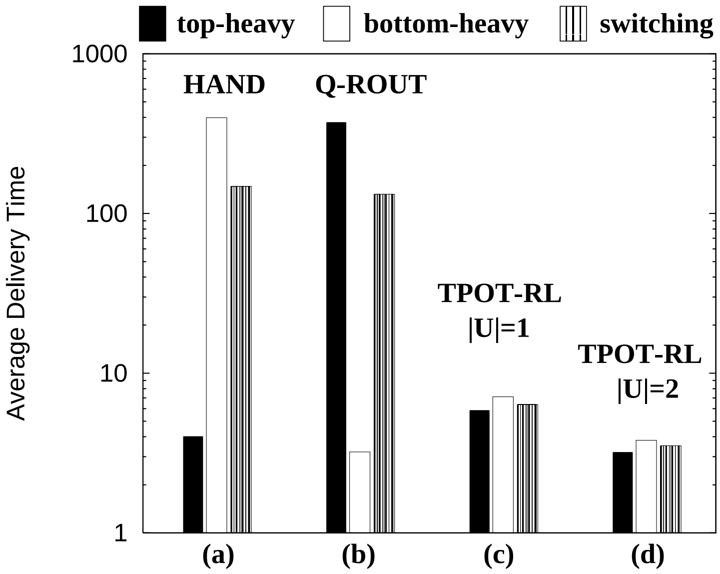

Figure 7.14: Fixed policies running in the top-heavy, bottom-heavy, and switching traffic patterns. The QROUT and TPOT-RL policies are both trained under switching conditions. Note the log scale on the y axis.

term reward. In this case, TPOT-RL is able to perform well under all traffic conditions with a single policy.

This last result repeats the result reported in the simulated robotic soccer domain in Section 7.3.4. Namely, TPOT-RL is capable of learning an effective policy under dynamic conditions in a team-partitioned, opaque-transition domain.

7.5 Discussion

As mentioned at the outset, this chapter serves a dual purpose. First, it represents the final layer in our layered learning implementation in the simulated robotic soccer domain. Second, it defines a new multi-agent learning technique that generalizes outside of the robotic soccer domain.

7.5.1 Pass Selection within Layered Learning

Pass selection represents the third and highest-level behavioral layer within our layered learning implementation. Its implementation follows the principles of layered learning as laid out in Section 4.1:

- Pass selection is a behavior that must be adaptable. Since it depends on the behaviors of teammates and opponents, agents must be able to adjust their decisions based on the empirical results of past decisions. Thus, pass selection is an appropriate behavior for learning because of the possibility of exploiting data to adapt to a shifting concept.

- We chose the machine learning method for pass selection based on the task characteristics. As presented in Section 7.1, no pre-existing machine learning method is suitable for learning pass selection in simulated robotic soccer. Therefore, we introduce a new multi-agent reinforcement learning algorithm, TPOT-RL, to accomplish this task.

  Low-level learning in complex domains can be well-defined with easy-to-isolate problems that allow agents to gather extensive training examples. However, when learning high-level multi-agent behaviors, training opportunities can be sparse and agents cannot be trained in isolation since agent policies are interdependent. Although RL typically suffers from the "curse of dimensionality," the use of TPOT-RL to learn a successful pass-selection behavior indicates that it can be used to learn complex multi-agent behaviors with few training examples in very large state spaces.

- The previous learned layer—pass evaluation—is used as part of the input space for TPOT-RL. In particular, the trained DT for pass evaluation is used to define the crucial action-evaluation function e when using TPOT-RL for learning pass selection. The previously-trained DT gives an indication of the likelihood that a given pass will succeed based on the configuration of teammates and opponents, but regardless of the receiver's position on the field. The trained RCF then maps the collection of feature values (decision tree classifications) to a Q-value for each action. A key assumption is that the Q-value of a particular action does not depend on the feature values for the other actions. The Q-values indicate the long-term strategic value of taking a particular action given the predicted short-term effect of that action in the current state.

In this chapter, TPOT-RL is used to learn pass selection, the third layer of our layered learning implementation. When learning pass selection, the team of agents uses a fixed formation against a fixed opponent. As presented in Section 4.2.2, our layered learning implementation could be extended by incorporating pass selection within higher-level learned behaviors. For example, the team might learn to adjust its formation against a fixed opponent. It might also learn to switch among pass selection policies when facing different opponents.

7.5.2 TPOT-RL

TPOT-RL is an effective technique for enabling a team of agents to learn to cooperate towards the achievement of a specific goal. It is an adaptation of traditional RL methods that is applicable in complex, non-stationary, multi-agent domains with large state spaces and limited training opportunities. TPOT-RL enables teams of agents to learn effective policies with very few training examples even in the face of a large state space with large amounts of hidden state.

In short, TPOT-RL applies in domains with the following characteristics:

- There are multiple agents organized in a team.

- There are opaque state transitions.

- There are too many states and/or not enough training examples for traditional RL techniques to work.
- The target concept is non-stationary.
- There is long-range reward available.
- There are action-dependent features available.

There are several such domains. As seen in Sections 7.3 and 7.4, simulated robotic soccer and network packet-routing are two such team-partitioned, opaque-transition domains. Other domains to which TPOT-RL could potentially be applied are information networks, distributed logistics, and rescue missions. For example, an information agent may broadcast a message without any knowledge of who will receive and react to it.

In all of these team-partitioned, opaque-transition domains, a team of agents works together towards a common goal, but each individual agent only executes a portion of the actions along the path to the goal. Agents can control their own destinies only intermittently and at irregular intervals. While not in full control of the team's goal achievement, they must still learn how to act so as to *help* their team achieve its goal. These domains are in contrast with, for example, grid world domains in which a single agent moves from some initial location to some final goal location; domains where agents take actions in parallel though also possibly in coordination—two robots executing tasks in parallel; and game domains where the rules of the game require that an agent and its opponent alternate actions.

One main contribution of this thesis is the adaptation of the RL paradigm to a non-stationary, opaque-transition multi-agent domain with a huge state space and extremely limited training examples. TPOT-RL succeeds in this challenging domain by:

- Partitioning the value function among multiple agents.
- Training all agents simultaneously with a gradually decreasing exploration rate.
- Using action-dependent features to aggressively generalize the state space.
- Gathering long-term, discounted reward directly from the environment.

7.6 Related Work

The use of machine learning in multi-agent systems has recently been receiving a good deal of attention. For a detailed discussion, see Chapter 9. This section highlights some of the prior RL research that is most related to TPOT-RL.

TPOT-RL mixes characteristics of different RL approaches. From a team perspective, there are a series of action opportunities resulting in state transitions and occasionally leading to rewards. In this sense, it is similar to Q-learning. However, from an individual agent's perspective, since transitions are opaque, TPOT-RL is more like a Monte Carlo system in which actions are rewarded for their eventual outcomes without any regard for the path of states traversed between the action and the reward. In this sense, TPOT-RL is similar to TD(1) [Sutton 88] [5].

One method of dealing with large state spaces in RL is approximation of the function Q. There has been much research into function approximation techniques in which the value function is a neural network or perhaps a decision tree (e.g. [Boyan and Moore 95]—see [Kaelbling *et al.* 96] for a survey). In contrast to TPOT-RL's construction of a small feature space prior to learning, function approximators generalize large state spaces during learning.

When using a feature space V, the definition of V can have a huge effect on the nature of Q. For example, in [Salustowicz *et al.* 98], a grid-like discretization is used for V. Since too many states result for a lookup-table, a neural network is used as the value function approximator. This approach is shown not to work very well, and the authors conclude that a more complex function approximator might work better. In contrast, we take the approach of using a smaller feature space and the simplest possible evaluation function: a lookup-table.

One RL application that has some similar characteristics to simulated robotic soccer is distributed elevator control [Crites and Barto 96]. In this domain, a team of RL agents—each responsible for one of four elevator cars—in a large, partially hidden, continuous state space. Since the four agents learn simultaneously, the reinforcement to each individual agent is non-stationary: it depends on the policies of the other agents. Crites and Barto report very good results in comparison with heuristic control policies.

Elevator control as presented in [Crites and Barto 96] differs in two main ways from the problems with which we are concerned. First, the approach is not team-partitioned since each agent learns to operate in the entire state space. Second, it is not opaque-transition. After taking an action in a state, an elevator agent knows the subsequent state that it enters and is able to update its value function based on the value of this subsequent state.

TPOT-RL is able to learn with very few training examples. Prioritized sweeping [Moore and Atkeson 93] is another method for learning with limited training examples. However, like many other RL techniques, it relies on being able to store trajectories of state transitions in the environment.

Even the partially observable Markov decision process (POMDP) [Kaelbling *et al.* 94] framework, which is designed for problems with hidden state, relies on agents having some knowledge of state transitions: POMDPs assume that the agent knows when the system has transitioned to a new state and a new action can be taken. Thus reward can still be propagated back through the state-action trajectory. Without access to future action opportunities agents must accumulate their rewards directly from the environment.

Although the use of the environment for reward accumulation is a necessity in this domain, it has also been shown to be an advantage in similar situations. Searching in stochastic policy space and using an average payoff for evaluating observation-action pairs can produce successful policies without relying on the Markov property [Singh *et al.* 94]. As in our case, the work reported there involves a domain in which several states are in the same observation class due to the use of coarse features.

[5] TD(1) is equivalent to a form of Monte Carlo learning [Singh and Sutton 96].

The intermediate reinforcement in the reward function R is similar to *progress estimators* [Mataric 94a]. Progress estimators use the short-term real-world effects of actions as intermediate rewards to help robots reach their ultimate goal location. Mataric's *conditions* also play a similar role to the features used here, reducing the size of the domain of the value function. This work was done using a reactive approach, as opposed to our more deliberative approach which takes into account past world states and agents' internal states.

Previous multi-agent reinforcement learning systems have typically dealt with much simpler tasks than the one presented here. Littman uses Markov games to learn stochastic policies in a very abstract version of 1-on-1 robotic soccer [Littman 94]. There have also been a number of studies of multi-agent reinforcement learning in the pursuit domain with four predators chasing a single prey in a small grid-like world. For example, Tan [93] compares situations in which predator agents are allowed to share reinforcement information and/or policies; Arai [97] provides agents with reinforcement for enabling successful actions by teammates; and Ono [97] equips each predator agent with different behavior modules based on how many teammates are closer than it is to the prey. Even the relatively complex backgammon [Tesauro 94] and elevator control [Crites and Barto 96] domains have much smaller state space than the simulated robotic soccer domain.

In another predator-like task, Zhao and Schmidhuber [96] use a single run to deal with the opponents' shifting policies and ignore the opponents' policies just as we do. The effects of opponent actions are captured in the reward function.

In robotic soccer, a reinforcement learning approach has been used for strategic positioning [Andou 98] in the soccer server. Introducing *observational reinforcement learning*, this system allows players to notice where the ball has traveled most often in the past and to adjust their positions such that they are closer to the ball's path in the future.

Within network routing, Q-routing [Boyan and Littman 94] is discussed in detail above as it is one of the benchmarks against which we compare TPOT-RL. Another approach to network routing in the same simulator is inspired by an ant metaphor [Subramanian *et al.* 97]. Ants crawl backwards over the network to discover link costs and shortest paths. This system rests on the assumption that link costs are the same for packets going in either direction. It also adds an overhead cost of sending the ant packets through the network.

Similarly inspired by the ant metaphor, AntNet [Caro and Dorigo 98] agents traverse a routing network and write information at the nodes reflecting their experience of the current network status. Within the framework presented in [Caro and Dorigo 98], TPOT-RL is a distributed, adaptive, non-minimal (i.e. packets do not necessarily always go along minimal cost paths), and optimal (i.e. the objective is to optimize the entire network's performance as opposed to any individual packet's traversal time) routing algorithm.