Bi-level Optimization for Learning Cost Functions from Demonstration

Chris Mills-Price, Weng-Keen Wong, Prasad Tadepalli and **Ethan Dereszynski**
School of Electrical Engineering and Computer Science
Oregon State University
{millspch,wong,tadepall,dereszet}@eecs.oregonstate.edu

Introduction

An effective way for a novice to learn a new complex task is to observe an expert demonstrate how the task should be accomplished. While the expert demonstration provides all the necessary information for solving that particular instance of the task, the novice needs to be able to generalize from the demonstration in order to accomplish similar tasks in different settings. One way for the novice to generalize to other situations is to learn the expert's preferences over goal configurations. In many domains, there may be multiple goal states for a given task. However, an expert prefers one goal state over other alternatives and thus demonstrates a particular sequence of steps leading to the preferred goal state. If each goal state is characterized as a vector of features, an internal cost function belonging to the expert can map the features values to a cost. Since the expert chose a particular goal state during the demonstration, we assume that this chosen goal configuration has an optimal cost with respect to the other alternatives considered by the expert.

Methodology

We now formulate this problem using a planning framework. Let the problem be defined as a 7-tuple $(S, A, T, s_I, G, E, \Psi)$ where S is a finite set of states, A is a finite set of actions, T is a deterministic transition matrix specifying the next state given the current state and an action, s_I is the initial state and G is the set of goal states. The expert provides a demonstration E which results in a set of state-action pairs starting at the initial state s_I and terminating at a particular goal state $g_E \epsilon G$. In general, the expert may provide more than one demonstration, hence E would consist of multiple trajectories from the start state to a goal state. We will assume a single demonstration to simplify the notation. If we imagine the problem in terms of a search tree, the state s_I is the root and each branch emanating from the initial state corresponds to an available action which leads to the next state. The branch grows downward until a goal state is reached. The expert demonstration forms a path through this search tree, terminating at the leaf node g_E. Finally, Ψ is a joint feature vector obtained over the start state s_I and a goal state g. Although we could use only the features of the goal state to determine its cost, we have often found it useful to derive features from both the initial state and the goal state. $\Psi(s_I, g)$ can be viewed as a function mapping the initial state and goal state to a vector of features. We abbreviate the joint feature vector as Ψ for convenience.

There is, however, one important detail regarding the action space in our formulation. The finite set of actions in A are permitted to have continuous parameters. For example, an action to build a town hall in a real-time strategy game domain could be represented as $BuildTownHall(x, y)$, in which the parameters x and y represent continuous values for the coordinates of the town hall. Thus, if we continue with our search tree view of the problem, each node of the search tree still has a finite number of actions available. However, if we leave all parameters uninstantiated until we reach a leaf node, each leaf node now represents an infinite set of goal states. The expert trace still forms a path through this tree. At the leaf node for the expert trace, the parameters for each action are instantiated with values observed from the demonstration.

The overall task is to learn a cost function that captures the expert's preferences over features of the goal state. We make the assumption that the expert's cost function is a weighted linear combination of the components of the joint feature vector Ψ. Specifically, the cost function can be written as $C(s_I, g, \mathbf{w}) = \langle \mathbf{w}, \Psi(s_I, g_E) \rangle$, where the angle brackets represent a dot product. Thus, the overall task can be summarized as learning the weights such that the cost function for the goal state chosen by the expert has the lowest cost among all other goal states ie. find $\mathbf{w}$ such that:

$$\langle \mathbf{w}, \Psi(s_I, g_E) \rangle < \langle \mathbf{w}, \Psi(s_I, g) \rangle \text{ for all } g \neq g_E. \quad (1)$$

There are numerous ways to assign weights so that Equation 1 holds. We assign values to the weights so as to maximize the margin between the joint feature vector belonging to the expert's goal state and all other joint feature vectors. In order to compute these max-margin weights, we can set up a quadratic program similar to that used in Support Vector Machines. However, there are an infinite number of constraints in this quadratic program. To see where these constraints come from, consider the goal states g in the right-hand side of Equation 1 which correspond to alternative goal states not chosen by the expert. We will refer to the goal state g_E as the *expert* goal state and the goal states g

as the *non-expert* goal states. Due to the fact that the actions are parameterized by continuous parameters, there are an infinite number of non-expert goal states. In the max-margin quadratic program, for each non-expert goal state, we need to add a constraint that requires that goal state's cost to be higher than the cost of the expert goal state.

In order to find the max-margin solution, we follow the approach taken by SVMStruct (Tsochantaridis *et al.* 2005), which computes an approximate max-margin solution by incrementally adding constraints. SVMStruct is an algorithm intended for structured classification. Structured classification, which is similar to multiclass learning, involves learning a classifier to predict an output variable that is a tuple. The components of this output tuple relate to each other in some way and form a structured output. For example, the output tuple could correspond to features of a goal state.

SVMStruct works by first finding the "best" non-expert goal state, which is the goal state with the lowest cost. If, under the current weights, the best non-expert goal state has a lower cost than the expert goal state, SVMStruct adds a constraint which requires the cost of the best non-expert goal state to be higher than the cost of the expert goal state. Then, the max-margin weights for the cost function are recalculated with respect to these constraints. This process continues until the best non-expert goal state has a lower cost than the expert goal state by an amount less than some epsilon. Note that this brief description of SVMStruct glosses over details such as the incorporation of slack variables and arbitrary loss functions into the quadratic program.

The inner loop of SVMStruct requires a step that finds the best non-expert goal state. This step is complicated by the fact that by allowing the actions to take continuous parameters, $\Psi(s_I, g)$ may have continuous valued features. Consequently, a search of the search tree in combination with a constrained optimization step is required to find the joint feature vector corresponding to the best non-expert goal state. If we take a higher level view of the procedure for learning the cost function, we see that each iteration of the main SVMStruct loop requires two optimizations. First, a constrained optimization step is needed to generate the "best" non-expert goal state. Secondly, an inverse optimization step is required to recompute the weights in the cost function. This two-stage optimization is a specific instance of a bi-level optimization problem. As a result, we name our algorithm the Bi-level Decision-Theoretic Learner (BL-DTL).

Evaluation

Building a well-structured base in a real-time strategy game such as Wargus is a challenging task that is similar to a classical floor-planning problem. We report preliminary results on this base-building task. In our experiments, the initial state consists of a map with a gold mine and a single tree. The set of actions include $BuildTownHall(x, y)$, $BuildLumberMill(x, y)$, $BuildTower(x, y)$ and $BuildFarm(x, y)$, where x and y are continuous coordinate values. A goal state consists of a map with a town hall, lumber mill, tower and farm. The features extracted from each goal state include the town hall to gold mine distance, the lumber mill to tree distance, the sum of the distances of the buildings to the enemy corner, and the base compactness. There are also a number of hard constraints that must be respected. The x and y locations of the buildings must be within the boundaries of the 32 by 32 map. There are also minimum distances between the town hall and mine as well as between the lumber mill and tree. Finally, buildings cannot be built on top of each other.

We produced the expert demonstration by first generating a known cost function. The expert goal state with respect to this cost function is determined by placing the four buildings in locations that minimize the cost. This goal state is approximately achieved through constrained gradient descent. We refer to the cost of this approximate goal state as the *near-optimal solution cost*. Once the map representing the expert demonstration has been produced, we run BL-DTL in a *learning phase* to learn the expert's cost function. The constrained optimization step in SVMStruct uses constrained gradient descent with random restarts. Once BL-DTL learns the weights for the cost function, it then moves to a *performance phase* where it operates on a previously unseen map and builds the desired list of buildings. BL-DTL uses a the same constrained gradient descent method as in the learning phase. We evaluate BL-DTL along with another algorithm called SVMStruct-Rand. In SVMStruct-Rand, the constrained optimization step which retrieves the best non-expert solution is replaced with a step that returns a random legal non-expert solution. We also run two versions of the base-building task. A simpler version involves building just two buildings – a town hall and a lumber mill and a simpler cost function that excludes base cohension and the distance to enemy. The full version involves building all four buildings and using all four features in the cost function.

For each algorithm, we report the difference between the cost of the final solution and the cost of the near-optimal solution averaged over 50 runs in Table 1. We also report 95% confidence intervals. As expected, BL-DTL dramatically outperforms SVMStruct-Rand on the simpler version of the problem. For the full problem, BL-DTL has a slight edge over SVMStruct-Rand, possibly due to the fact that the hard constraints make the search space so fragmented that the gradient descent step has little improvement over the random selection step in SVMStruct-Rand.

	BL-DTL	**SVMStruct-Rand**
2 buildings	2.020 ± 2.13	130.90 ± 59.68
4 buildings	27.69 ± 15.31	30.80 ± 11.65

Table 1: 95% confidence interval for the average difference between final solution cost and near-optimal solution cost

Acknowledgements

This research was funded by DARPA under contract FA8650-06-C-7605.

References

Tsochantaridis, I.; Joachims, T.; Hofmann, T.; and Altun, Y. 2005. Large margin methods for structured and interdependent output variables. *Journal of Machine Learning Research* 6:1453–1484.