Learning High-level Representations

James MacGlashan

1 Introduction

The purpose of this document is to propose some general approaches for learning high-level representations of a planning problem that can then be exploited by planning algorithms. We will assume that the state is fully observable and we will also assume that in addition to the state, some additional perception-based features will be provided (e.g., vision, a map, etc.). The agent will be provided a training period in n different planning problems in which planning without high-level information is tractable. The outputs of the training that we'd like are (1) a set of high-level features that are determined from the perception in each state; a set of subgoals, similarly defined in learned high-level features; and a set of affordances associated with each subgoal. For simplicity, we will break the core ideas into related but simpler problems that we can test independently.

2 Learning Problem-level Affordances

Let us start by considering a simpler problem of learning which classes of planning problems exist in a set for training problems, which action distributions the agent should use in each to bias planning, and how to identify the problem class from perception. We will define a planning problem as a pair (M, s_0), where M is an MDP definition and s_0 is an initial state. A problem class defines a set of problems for which there is a common action distribution that would be used in planning. For instance, the flat world class in Minecraft would only require using the movement actions, whereas a trench world class would require the use of movement and placing blocks. Finally, we will define $P \times S \to \mathbb{R}^m$ to be a function that produces the perception features available to the agent.

The most trivial form of this problem is to imagine that the agent will only ever need to plan for the same planning problem. This problem is trivial because we do not need to identify which problem class the agent is ever in, only what action distribution would be used if the agent ever had to replan. We can imagine this simplified problem as if the agent will have to forget the policy it planned during training, but can remember the action distribution so that it can trivially replan the problem again in the future. Another way to think of the result is as a lossy compression of the policy.

We can produce the corresponding action distribution for a single problem using Algorithm 1, which finds a policy for the problem, computes the frequency of each action in the set of states that are reachable from the initial state, and returns a probability for each action by normalizing.

Algorithm 1 LearnActionDistribution(M, s_0)

```
Input: MDP $M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, initial state $s_0$
  $\pi \leftarrow$ Plan($M, s_0$)
  $\hat{\mathcal{S}} \leftarrow$ Reachable($s_0, \pi$)        ▷ All reachable states from $s_0$ when following $\pi$
  $c(a) \leftarrow |\{s \in \hat{\mathcal{S}} : \pi(s) = a\}| \ \forall a \in \mathcal{A}$        ▷ count action frequency
  $\Pr(a) \leftarrow \frac{c(a)}{\sum_{a' \in \mathcal{A}} c(a')}$
  return $\Pr(a), \pi$
```

A slightly more relaxed version of this problem is if we had a set of problems that were known to be from the same class. In this case, we generate an action distribution for the class by averaging the the result of Algorithm 1 applied to each of them. (Alternatively, if the set of reachable states in each problem under its policy can vary significantly in size between problems, the sum of the action frequencies for each problem in the class can be added together and then normalized.)

If a set of training problems from different labeled problem classes was available, but the agent could not expect to the know the problem class outside of training, then a supervised learning algorithm can be used to learn a mapping from perception to the problem class, after which the agent can use the associated action distribution that was learned for each problem class to bias planning. Specifically, Algorithm 2 shows how to setup the supervised learning task, where each training problem contributes a supervised learning training data instance for each state that is reachable from following the solved policy from the problem's initial state. The supervised training instance for each state is the perceptual features for the state with the problem class as the supervised label. After the supervised training dataset is constructed, any suitable off-the-shelf supervised learning algorithm can be used to train a classifier, which is then returned.

Algorithm 2 LearnPerceptionToProblemClass($n, \mathcal{P}, \Pi, \mathcal{C}$)

```
Input: $n$ problems $\mathcal{P}$, solved policies $\Pi$, and associated problem classes $\mathcal{C}$
  $D \leftarrow \{\}$
  for $i = 1$ to $n$ do
    $\hat{\mathcal{S}} \leftarrow$ Reachable($\mathcal{P}_i.s_0, \Pi_i$)
    for all $s \in \hat{\mathcal{S}}$ do
      $D \leftarrow D \cup \{(P(s), \mathcal{C}_i)\}$
    end for
  end for
  $c \leftarrow$ LearnClassifier($D$)
  return c
```

We now return to the question of when the problem class is not provided to the agent in training problems and instead must be identified by the agent in an unsupervised manner. Recall that we are defining a problem class as a set of problems that we would expect to require the same kinds of action distributions to solve. Therefore, identifying problem classes from unsupervised training problems is accomplished by generating the action distributions for each problem (using Algorithm 1) and then clustering the problems in their action distribution space, using a metric such as the Kullback-Leibler divergence or the Fisher information metric. The clustering algorithm should ideally pick the number of clusters from the data.

After clustering is performed, the cluster to which each problem is assigned is used as the problem class label, after which Algorithm 2 is used to identify the problem class from perceptual features. Algorithm 3 shows the final algorithm that identifies problem classes from training data, generates action distributions for each, and learns how to identify them from perceptual features to bias planning in novel target problems.

```
Algorithm 3 IdentifyAndLearnProblemClasses($n, \mathcal{P}$)
Input: $n$ and the training problems $\mathcal{P}$
  for $i = 1$ to $n$ do
    $A_i \Pi_i \leftarrow$ LearnActionDistribution($\mathcal{P}_i.M, \mathcal{P}_i.s_0$)
  end for
  $K, k \leftarrow$ Cluster($A$)      ▷ cluster assignment function ($K$) and # of clusters $k$
  for $i = 1$ to $k$ do
    $\mathcal{A}_i \leftarrow$ Average($\{a \in A | K(a) = i\}$)      ▷ Average for each cluster
  end for
  for $i = 1$ to $n$ do
    $\mathcal{C}_i \leftarrow K(A_i)$
  end for
  $C \leftarrow$ LearnPerceptionToProblemClass($n, \mathcal{P}, \Pi, \mathcal{C}$)
  return $C, A$
```

3 Features, Subgoals, and High-level MDPs

Although Algorithm 3 should be useful in scaled up versions of the training tasks, it may be less useful when the agent has to plan in complex worlds that require many different subgoals. However, using similar approaches, we may be able to also learn subgoals, high-level features to represent them, and then create high-level subgoal achieving "actions" for them, all from which a high-level MDP can be generated and used to accelerate planning in complex tasks.

Ideas to be filled in later:

- cluster by time in a single problem; score clusters by action distribution changes.

- supervised learning produces features that predict the existence of different sub-problem action distribution changes and used as subgoal features.

4 Research Plan

I believe that Algorithm 1 is effectively already implemented in the current affordance work, or at least something very similar to it. Therefore, I suggest that the next task is to implement the clustering part of Algorithm 3 and we should examine which clusters it finds and how it assigns problems to them. We should consider it a success if we see problems like flat worlds in one cluster and problems like bridge world in another. If so, then we should consider the supervised learning problem and see if we can provide perception information (e.g., a 2D map image) from which supervised learning can effectively learn to identify the correct cluster.