Supplementary Material for Learning Heuristic Search via Imitation

Mohak Bhardwaj
The Robotics Institute
Carnegie Mellon University
mohakbhardwaj@cmu.edu

Sanjiban Choudhury
The Robotics Institute
Carnegie Mellon University
sanjiban@cmu.edu

Sebastian Scherer
The Robotics Institute
Carnegie Mellon University
basti@cs.cmu.edu

Experimental Setup

The initial phase of experimentation was carried out on a custom environment for 2D navigation planning compatible with OpenAI Gym [3]. The intention behind development of this environment was to easily benchmark the performance of state-of-the-art Reinforcement Learning and Imitation Learning algorithms on learning search heuristics.

In order to overcome the changing sizes of the observation and action spaces in our setting, we use insight from motion planning literature and represent an entire search state in terms of closest nodes in $\mathcal{O}$ to a set of pre-defined *attractor states* and *attractor paths*. Attractor states are manually defined states that can be thought of as landmarks trying to pull the search cloud in different directions. Such states can be useful in pulling the search out of local minima such as a bugtrap or they could be strategic orientations of the robot or an object the robot is trying to manipulate that lead to faster solutions [2]. Attractor paths on the other hand are solutions to a small subset of environments from the training dataset. In many episodic tasks, where the structure of the environment does not change drastically between planning iterations, such *path-reuse* can be very useful in finding solutions faster [5]. The planning algorithm is built into the environment, and the agent only receives as an observation the nodes in the open list closest to each attractor paths/states. At each iteration then, the action that the agent performs is to select a node from the observation to expand.

Although this is a generic framework that can be applied to many different problems, we chose not to use it for this work. The reason for this choice was that in this paper, our aim was to build the foundation for learning graph search heuristics as sequential decision making problem and clearly demonstrate the efficacy of the imitation learning paradigm in this domain. We found that using attractor paths/states would distract from the effectiveness of SAIL and also make learning easier for other baseline methods.

In our final experiments, we instead featurize every pair (v, s) using simple information based on the search tree and the environment uncovered up until that point. We talk more about the features in a later subsection.

Additionally, we have developed a simple and intuitive Python based planning pipeline to serve as a backend for the Gym environment. The planning environment makes it easy to incorporate machine learning libraries [7] [1] with custom planning graphs requires only environment images as input. We use this planning pipeline to conduct all our experiments. Source code and instructions can be found via our project page at this link: https://goo.gl/YXkQAC

Feature Representation

We explore different ways to construct representative features for a node in the search tree. Although technically, the features for a vertex v should depend on the parent edge e that leads to the vertex, we ignore this in practice and consider a vertex in isolation to calculate features. It is important to note that the features used must be easy to calculate (no high computational burden) and should only require information uncovered by search until that point in time(else it would count as extra expansions). To this end we divide our feature into two categories as follows:

Search Based Features: $f_S(v, s) = \left[x_v, y_v, g_v, h_{\text{EUC}}, h_{\text{MAN}}, d_{TREE}, x_{v_g}, y_{v_g}\right]$, where:

1st Conference on Robot Learning (CoRL 2017), Mountain View, United States.

- (x_v, y_v) - location of node in coordinate axis of occupancy map.
- (x_{v_g}, y_{v_g}) - location of goal in coordinate axis of occupancy map.
- g_v - cost(number of expansions) of shortest path to start.
- h_{EUC} - Euclidean distance to goal.
- h_{MAN} - Euclidean distance to goal.
- d_{TREE} - Depth of v in the search tree so far.

Environment Based Features: These features depend upon the environment uncovered so far, more specifically the vertices in $\mathcal{I}$.
$f_E(v, s) = [x_{OBS}, y_{OBS}, d_{OBSX}, x_{OBSX}, y_{OBSX}, d_{OBSX}, x_{COBSY}, y_{OBSY}, d_{OBSY}]$, where:

- $x_{OBS}, y_{OBS}, d_{OBSX}$ - coordinates and distance of closest node in $\mathcal{I}$ to v
- $x_{OBSX}, y_{OBSX}, d_{OBSX}$ - coordinates and distance of closest node in $\mathcal{I}$ to v in terms of x-coordinate.
- $x_{OBSY}, y_{OBSY}, d_{OBSY}$ - coordinates and distance of closest node in $\mathcal{I}$ to v in terms of y-coordinate

The net feature vector is then a concatenation of the two vectors i.e, $f = [f_S, f_E]$ which gives a 17-dimensional feature vector.

We also explored the possibility of using a local image patch representing the explored obstacles in black and free space in white as f_E, but found that the patch used must be large enough to contain useful information. Testing the efficacy of the image patch as feature and combining it with different machine learning models (eg. CNNs) would require further experimentation especially the computational trade-offs it poses.

Network Architecture and Hyperparameters

The function Q_θ is represented using feed-forward neural network with two fully connected hidden layers containing [100, 50] units and ReLu activation. The model takes as input a feature vector $f \in \mathcal{F}$ for the pair (v, s) and outputs a scalar cost-to-go estimate. The network is optimized using RMSProp [8]. A mini-batch size of 64 and a base learning rate of 0.01 is used. The network architecture and hyper-parameters are kept constant across all environments.

Practical Algorithm Implementation

Since the size of the action space changes as more states are expanded, the SAIL algorithm requires a forward pass through the model for every action individually unlike the usual practice of using a network that outputs cost-to-go estimate for all actions in one pass as in [4]. This can get computationally demanding as the search progresses ($\mathcal{O}(N)$ in actions). Instead, we use a *priority queue* as $\mathcal{O}$ which sorts vertices in increasing order of the cost-to-go estimates as is usually done in search based motion planning. The vertex on the top of the list is then expanded. We use two priority queues, sorted by the learner and oracle's cost-to-go estimates respectively. This allows us to take actions in $\mathcal{O}(1)$ but forces us to freeze the *Q*-value for a vertex to whenever it is inserted in $\mathcal{O}$. Despite this artificial restriction over the policy class Π, we are able to learn efficient policies in practice. However, we wish to relax this requirement in future work.

Details of Baseline Algorithms

Supervised Learning (Behavior Cloning)
The supervised learning algorithm is identical to SAIL with the key difference that roll-outs are made with π_{OR} and not π_{mix}. This is equivalent to setting the mixing parameter $\beta = 1$ across all environments. For completeness, we present the algorithm below in Alg. 1 We use $m = 600$ for all the environments. The network architecture and hyper-parameters used are the same as described in the prior subsection.

Algorithm 1: Supervised Learning

Initialize $\mathcal{D} \leftarrow \emptyset$,
Collect datapoints as follows:
for $i = 1, \ldots, m$ **do**
  Initialize sub-dataset $\mathcal{D}_i \leftarrow \emptyset$;
  Sample $\phi \sim P(\phi)$;
  Sample $(v_s, v_g) \sim P(v_s, v_g)$;
  Invoke clairvoyant oracle planner to compute $Q^{\text{COR}}(v, \phi) \, \forall v \in V$;
  Rollout a new search with π_{OR};
  At each timestep t pick a random action a_t to get corresponding (v, s_t);
  Query oracle for $Q^{\text{COR}}(v, \phi)$; ▷ `Look-up optimal cost-to-go`
  $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \langle v, s_t, Q^{\text{COR}}(v, \phi) \rangle$;
  Continue roll-out with π_{OR} till end of episode.;
Append to c.s classification data $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$;
Train $\hat{\theta}$ on $\mathcal{D}$ to get $\hat{\pi}$;
return $\hat{\pi}$

Q Learning with Function Approximation
We use an episodic implementation of the Q-learning algorithm which collects data in an iteration-wise manner similar to SAIL. Q_θ is trained on the aggregated dataset across all iterations by regressing to the TD-error. The aggregated dataset $\mathcal{D}$ effectively acts as an experience replay buffer to which helps in stabilizing learning when using neural network function approximation as has been suggested in recent work[4]. However,we do not use a target network or any other extensions over the original qlearning algorithm in our baselines, [9][6]. We also use only a single observation to take decisions and not a history length of past *h* observations for a fair comparison with SAIL which also uses a single observation. Alg. 2 describes the training procedure for the Q-learning baseline. C is the one step cost which is 1 for every expansion till goal is added to the open list. We use

Algorithm 2: Q-Learning

Initialize $\mathcal{D} \leftarrow \emptyset$, $\hat{\pi}_1$ to any policy in Π
for $i = 1, \ldots, N$ **do**
  Initialize sub-dataset $\mathcal{D}_i \leftarrow \emptyset$
  Let mixture policy be $\pi_{\text{mix}} = \epsilon$-greedy on $\hat{\pi}_i$ with ϵ_i
  Collect *mk* datapoints as follows:
  for $j = 1, \ldots, m$ **do**
    Sample $\phi \sim P(\phi)$;
    Sample $(v_s, v_g) \sim P(v_s, v_g)$;
    Sample uniformly *k* timesteps $\{t_1, t_2, \ldots, t_k\}$ where each $t_i \in \{1, \ldots, T\}$;
    Rollout a new search with π_{mix};
    At each $t \in \{t_1, t_2, \ldots, t_k\}$, $\mathcal{D}_i \leftarrow \mathcal{D}_i \cup \langle v, s_t, C, v_{t+1} \rangle$; ▷ v_{t+1} `is the least-Q action after executing` a_t `in` s_t
    Continue roll-out with π_{mix} till end of episode.;
  Append to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_i$;
  Train $\hat{\theta}_{i+1}$ by minimizing T.D error on $\mathcal{D}$ to get $\hat{\pi}_{i+1}$;
return *Best* $\hat{\pi}$ *on validation*;

$k = 100$ and $\epsilon_0 = 0.9$. Epsilon is decayed after every iteration in an exponential manner. Network architecture and params are kept the same as SAIL.

Cross Entropy Method (C.E.M) We use C.E.M as a derivative free optimization method for training $Q_{\hat{\theta}}$. At each iteration of the algorithm we sample $batch_{size} = 40$ set of parameters from a Gaussian Distribution. Each parameter is used to roll-out a policy on 5 environments each and the total cost is collected. The total cost (number of expansions) is used as the fitness function and the the best performing, $n_{elite} = 20\%$ of the parameters are selected. These elite parameters are

then used to create a new Gaussian distribution (using sample mean and standard deviation) for the next iteration. At the end of all iterations, the best performing policy on a set of held-out states is returned. For this baseline, we use a simpler neural network architecture with one hidden layer of 100 units and ReLu activation.

Analyzing the time complexity of SAIL

The computational bottleneck in SAIL is the `Select` function which requires estimating the Q-value for every node in the open list $\mathcal{O}$. Contrast this with something like Dijkstra's algorithm which selects a node to expand in very little time, but wastes a lot of computation in excessively expanding nodes and evaluating edges. In order to analyze the usefulness of SAIL in terms of computational gains, we make the following assumptions. Firstly, we assume that the computational cost of calculating the Q-value of a single node(including feature calculation and forward pass through function approximator) is equal to the computational cost of `Expand` function (involves checking all edges coming out of a node for collision and calculating edge costs). This is in reality a very conservative approximation as in many high-dimensional planning problems, collision checking is way more computationally demanding as it requires expensive geometric intersection computations. We also ignore the computational cost of re-ordering the priority-queue whenever a node is popped which means Dijkstra's algorithm can select a node to expand in $O(1)$. Given a graph with cardinality k, we obtain the following time complexities for SAIL and Dijkstra's algorithm.

SAIL test time complexity: Assume SAIL did A expansions before it found a solution starting from an empty $\mathcal{O}$. Also assume that states are never removed from $\mathcal{O}$.
Total `Select` complexity: $O\left(k + 2k + 3k + \ldots + Ak\right) = O\left(kA^2\right)$
Total expansion complexity: $O(\sum_{i=1}^{A} k) = O(Ak)$
From this we get total complexity of SAIL to be $O\left(kA^2\right)$.
Dijkstra's test time complexity: Assume Dijkstra's algorithm does B expansions before finding a path. As mentioned earlier, we assume that priority-queue reordering can be achieved in constant time.
Total `Select` complexity: $O(1)$
Total expansion complexity: $O\left(kB\right)$ From this we obtain total complexity for Dijkstra's algorithm to be $O\left(kB\right)$.

From the above analysis, for SAIL to have lesser overall computational complexity than uninformed search we require the following condition to be satisfied:

$$A^2 < B \tag{1}$$

Thus, SAIL must obtain a squared reduction in total number of expansions for it to be computationally better than uninformed search. We argue that this strengthens the case for using SAIL in higher dimensional search graphs as in uninformed search expands a very large number of nodes as the total number of graph nodes increases.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. URL http://arxiv.org/abs/1603.04467.

[2] S. Aine, S. Swaminathan, V. Narayanan, V. Hwang, and M. Likhachev. Multi-heuristic a. *The International Journal of Robotics Research*, 2016.

[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym, 2016.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

[5] M. Phillips, B. J. Cohen, S. Chitta, and M. Likhachev. E-graphs: Bootstrapping planning with experience graphs. 2012.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015. URL http://arxiv.org/abs/1511.05952.

[7] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

[8] T. Tielman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[9] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. URL http://arxiv.org/abs/1509.06461.