Imitation-Net: A Supervised Learning Planner

Alan Fern, Murugeswari Issakkimuthu and Prasad Tadepalli
School of EECS, Oregon State University
Corvallis, OR 97331, USA

Abstract

Imitation-Net is an offline planner based on supervised learning to imitate an expert policy for the problem. It works in two phases - Training and Evaluation. During the training phase it creates a training dataset containing trajectories from an expert policy and trains a deep neural network (Goodfellow, Bengio, and Courville 2016) to approximate that policy. During the evaluation phase it runs the trained network to get an action for the current state by means of a single feed-forward pass.

Introduction

Figure 1 shows the schematic diagram of the entire planning system. There are three components: *RDDLSim* (Sanner 2010), the Java *RDDL* server used for evaluation in the competition, a C++ dynamic library based on *Prost* (Keller and Eyerich 2012), and a Python component consisting of a Tensor Flow policy network. *Prost* is the state-of-the-art search-based online planner for *RDDL* domains. The C++ dynamic library based on *Prost* acts as an intermediate layer between the *RDDL* server and the Tensor Flow network providing routines for communicating with the server, running the evaluation loop and simulating trajectories during the training phase.

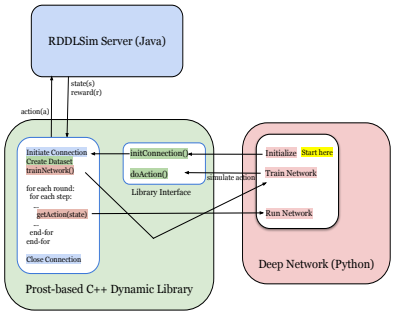

Figure 1: Schematic Diagram

The Python component starts the control flow by calling *initConnection()* in the dynamic library sending handles of callback functions *train()* and *test()* for training and running the policy network respectively. The dynamic library actually initiates (and also terminates) the communication with the server, receives and parses the *RDDL* domain and problem files, initializes the required data structures, and starts the network training process by invoking the *train()* callback function. The nested *for* loops in the dynamic library denote the evaluation loop in which it returns an action for the current state to the server and receives the reward and next state from the server. At each planning step the library invokes the *test()* callback function to run the policy network with the current state s and returns the received action $\hat{a}$ to the server.

The Policy Network

Architecture. The policy network is a sparsely connected structure with three hidden layers. The input layer has as many nodes as the number of state-fluents (n) in the problem, the hidden layers have $C * n$ nodes, and the output layer has as many nodes as the number of action-fluents in the problem (m). The hidden layers have *ReLU* non-linear units. The output layer has *sigmoid* non-linear units for the action nodes, thereby supporting factored action spaces directly. The network is not fully connected except at the final layer and connections going into the hidden layers are customized for each problem based on the transition function for state-fluents in the *RDDL* description. The input nodes represent state-fluents and the nodes in the hidden layers are clustered into n groups of C nodes each with each group representing a state-fluent. A hidden node corresponding to a state-fluent receives connections only from its parent nodes (in the previous layer) according to the state-fluent transition dynamics very much resembling a Dynamic Bayesian Network *(DBN)* structure. Figure 2 shows a sparse network for a problem with 4 state-fluents and 5 action-fluents for $C = 1$ and figure 3 roughly shows the same network for $C = 5$. The parameter C is called *channels* in the same sense as in convolutional neural networks. Reference (Issakkimuthu, Fern, and Tadepalli 2018) has more details on this sparse architecture.

Training Data Generation As shown in the schematic diagram in figure 1 the procedure for training data generation is kept within the C++ dynamic library mainly because it uses

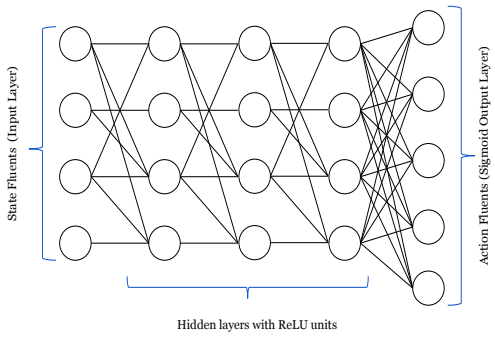

Figure 2: Network Architecture with one channel

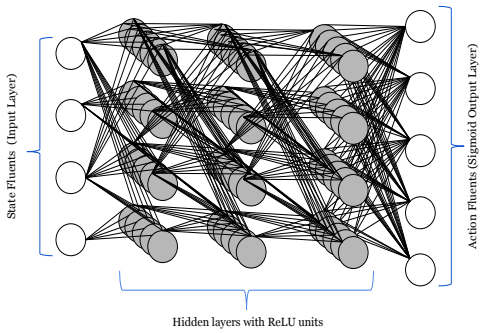

Figure 3: Network Architecture with 5 channels

some of the functionalities already available in *Prost*. The expert policy to imitate is the *Rollout-of-Random* policy, which is a one-step greedy policy over the value function of the random policy, i.e., the policy next in sequence to the random policy in the regular policy iteration sequence. Trajectories of the *Rollout-of-Random* policy are generated by estimating $Q^{\pi}(s, a)$ for each action $a \in A_s$ at state s for the random policy π and taking the best action to move to the next state s' from which the process is repeated. Therefore, states in the training set can be said to follow the state distribution of *Rollout-of-Random*.

Training The network is written in Tensor Flow and trained using Stochastic Gradient Descent *(SGD)* with a batch size of 10 to minimize the cross-entropy loss using the built-in *Adam* optimizer. During evaluation the probabilities of action-fluents computed by the trained network are used to compute the probabilities of individual applicable actions and the one with the highest probability is selected.

Implementation Details

The Python - C++ interface is implemented using the *ctypes* library (https://docs.python.org/3/library/ctypes.html). The important functionalities in *Prost* used in the dynamic library are

1. The *IPPCClient* class for establishing (and terminating) the connection with the *RDDL* server, parsing the *RDDL* domain and problem files and initializing data structures, and running the evaluation loop receiving state and reward signals and sending actions
2. The *RandomWalk* class for simulating a trajectory from state s starting with action a and then following the random policy π for h steps accounting for steps 12 through 19 in algorithm **??**
3. The *IDS* class to estimate the best rollout horizon h for the problem by means of iterative deepening search

Parameter Settings:

1. The competition imposes a *RAM* limit of 4GB for the planner. *RAM* usage is periodically monitored in the C++ function that creates the training dataset and the function is terminated once a limit of 2.5GB is reached. *RAM* usage is also monitored in the Python program while training the network and the training process is terminated once a limit of 3.5GB is reached.
2. The maximum number of training records in the dataset is limited to 30000, since larger datasets cannot be processed in the training process in limited time in the competition setting. The rollout horizon h for training data generation is initialized to the minimum of 5 or the value returned by the *IDS* class.
3. The total time (T) available to solve a problem instance needs to be divided between the training and evaluation phases leaving enough time for other associated computations like the initial parsing process. Approximately 70% of the total time T is set aside for just training the network. To be precise, an untrained network is run for one-fifth (15 for the competition) of the total number of episodes (75 in the competition) to compute a time t and time for final evaluation (t_e) is set to $2 \times t$ times the total number of episodes and time for data generation and training is set to 80% of $T - T_e$ out of which 30% is alloted for data generation and 70% is alloted for training.

Acknowledgements

Many thanks to Dr. Thomas Keller for his help with resolving problems connected to *Prost* functionalities.

References

[Goodfellow, Bengio, and Courville 2016] Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep learning*. MIT press.

[Issakkimuthu, Fern, and Tadepalli 2018] Issakkimuthu, M.; Fern, A.; and Tadepalli, P. 2018. Training Deep Reactive Policies for Probabilistic Planning Problems.

[Keller and Eyerich 2012] Keller, T., and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

[Sanner 2010] Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language description.