A2C-Plan: A Reinforcement Learning Planner

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

Abstract

A2C-Plan is an offline planner that trains a policy network through Reinforcement Learning (RL) using an Advantage Actor-Critic (A2C) algorithm. It works in two phases - Training and Evaluation. In the training phase it trains a deep neural network (Goodfellow, Bengio, and Courville 2016) for the problem instance using the A2C algorithm with simulated trajectories and normalized rewards. In the evaluation phase it uses 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 PyTorch 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 PyTorch network providing routines for communicating with the server, running the evaluation loop and simulating trajectories during the training phase.

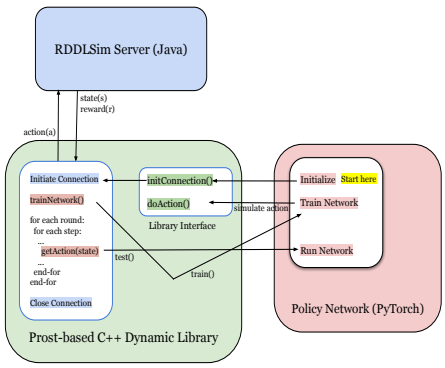

Figure 1: Schematic Diagram

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.

Training the Policy Network

The policy network is a fully-connected network with two hidden layers. The input layer has as many nodes as the number of state-fluents (n) in the problem, the hidden layers have $3 * n$ and $2 * n$ units respectively, and the final layer has as many action nodes as the number of ground actions (m) in the problem plus an additional value node. The hidden layers have *ReLU* non-linear units and the output layer is a *softmax* layer that computes a probability distribution over the set of m actions. Figure 2 shows the network architecture for a problem with 2 state-fluents and 4 ground actions.

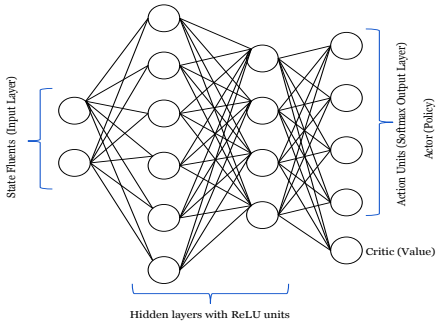

Figure 2: Network Architecture

Actor-Critic Algorithms

Actor-Critic methods (Konda and Tsitsiklis 2000) bring together the advantages of actor-only methods that directly learn a parameterized policy and critic-only methods that learn a value function by means of training a critic network using simulations and using the learned critic values to make gradient updates to the parameters of the policy network. In *A2C-Plan* the actor and critic share the same network up to the penultimate layer as shown in figure 2. The Advantage Actor-Critic (A2C) algorithm uses the *Q-Advantage* of an action $(Q(s,a) - V(s))$ instead of the value of a state $V(s)$ to update the actor parameters.

Algorithm 1 below is just meant for outlining the steps involved for one training episode. Details of the Generalized Advantage Estimation (GAE) procedure used in the implementation for updating *actor-loss* can be found in (Schulman et al. 2016). The functions *critic(s)* and *actor(s)* return the critic value and actor probabilities computed by the network respectively, *reset(env)* resets the environment and returns the initial state of an episode, *sample(P)* returns an action sampled using the probability distribution P computed by the actor network along with the probability p_a of the selected action a, and *max* and *min* are the maximum and minimum reward values for the problem computed or approximated by *Prost*. The entropy term e in the actor loss function encourages exploration.

Algorithm 1 A2C(*net*, *env*)

repeat
  Create Arrays V, R, L, E
  Initialize $i \leftarrow 0$, $s \leftarrow$ reset(*env*)
  while not end-of-episode **do**
    $v \leftarrow$ *critic*(s), $P \leftarrow$ *actor*(s)
    $(a, p_a) \leftarrow$ sample(P)
    $l \leftarrow log(p_a)$
    $e \leftarrow \sum_{p_a \in P} log(p_a)$
    $(s', r) \leftarrow$ next-state(s, a)
    $r \leftarrow r/(max - min)$
    $V[i] \leftarrow v$, $R[i] \leftarrow r$
    $L[i] \leftarrow l$, $E[i] \leftarrow e$
    $s \leftarrow s'$, $i \leftarrow i + 1$
  end while
  critic-loss $\leftarrow$ *actor-loss* $\leftarrow 0$
  $\hat{v} \leftarrow 0$
  for $i = H \ldots 1$ **do**
    $\hat{v} \leftarrow \gamma\hat{v} + R[i]$
    critic-loss $\leftarrow$ *critic-loss* + $(V[i] - \hat{v})^2$
    Adv $\leftarrow R[i] + \gamma V[i+1].value - V[i].value$
    actor-loss $\leftarrow$ *actor-loss* - *Adv* * $L[i] + E[i]$
  end for
  Minimize *critic-loss*, *actor-loss*
until time-limit or memory-limit is not reached

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 *SearchEngine* class functions estimating the maximum and minimum rewards for the problem instance
3. All the classes involved in simulating an action at a given state to compute the reward and the next state

Parameter Settings:

1. The competition imposes a *RAM* limit of 4GB for the process. *RAM* usage is periodically 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 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 training is set to 75% of $T - T_e$.
3. For domains with action pre-conditions some of the actions might not be applicable at a given state. When that happens during training or evaluation an applicable action with the highest probability is used instead.

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.

[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*.

[Konda and Tsitsiklis 2000] Konda, V., and Tsitsiklis, J. 2000. Actor-critic algorithms. In *SIAM Journal on Control and Optimization*, 1008–1014. MIT Press.

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

[Schulman et al. 2016] Schulman, J.; Moritz, P.; Levine, S.; Jordan, M.; and Abbeel, P. 2016. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations (ICLR)*.