Action Schema Networks – IPC Version

Mingyu Hao,[1] **Sam Toyer,**[2] **Ryan Wang,**[1] **Sylvie Thiébaux,**[1] **Felipe Trevizan**[1]

[1]School of Computing, The Australian National University
[2]Department of Electrical Engineering and Computer Sciences, University of California, Berkeley
mingyu.hao@anu.edu.au, sdt@berkeley.edu, ryan.wang@anu.edu.au, sylvie.thiebaux@anu.edu.au,
felipe.trevizan@anu.edu.au

Abstract

Action Schema Networks (ASNets) are neural network architectures for generalized reactive policies that leverage the relational nature of planning problems represented in (P)PDDL. An ASNet policy can be learned by imitating the actions selected by a traditional non-learning planner. Even when the training set consists of only a few small problems, ASNets are able to solve significantly larger instances within the same domain. In this paper, we describe the ASNet planner submitted to the Learning Track of the International Planning Competition (IPC23). It closely follows the original implementation (Toyer et al. 2020), but focuses on solving deterministic planning problems and employs newer PDDL parsers and eager-execution-based machine learning frameworks.

1 ASNets

In this section we briefly describe the core ideas of ASNets. For a complete and comprehensive explanation of the model we refer the reader to (Toyer et al. 2020). Action Schema Networks (ASNets) attempt to connect the world of automated planning to deep learning (Toyer et al. 2018) by encoding the relational structure of factored planning problems into neural networks, where neurons represent actions and propositions. Connections between each layer are derived from a graph representation based on the action schema of the given domain. Because all problems from the same domain share the same action schema, weights of the policy network trained on a small set of problems can be shared with networks instantiated for larger problems from the domain. Therefore, training with simpler problems makes solving harder problems with policy networks possible using the learned parameters.

ASNets are policy networks that take in the current state s and return a policy $\pi^\theta(a|s)$. The current state is encoded as a feature vector and fed into an alternating sequence of action and proposition layers. Each action layer consists of action modules representing actions, and each proposition layer is formed by proposition modules corresponding to propositions. The connections between modules are determined by the $relatedness$ of actions and propositions.

Definition 1 ($Relatedness$**)** *A proposition p is related to an action a at position k if the lifted proposition (predicate) is the k-th unique predicate in the precondition or effect of the lifted action (action schema).*

A proposition module is connected to an action module in the next layer, if and only if the underlying action and proposition are related. Similarly, action modules are connected to related proposition modules in the next layer. Because the relatedness only depends on predicates and action schemas, propositions and actions grounded from the same action schema share the same number of connections.

Each action module at layer l with underlying action a takes in a vector $u_a^l \in \mathbb{R}^{d_a^l}$ and outputs another vector $\phi_a^l \in \mathbb{R}^{d_h}$, where d_h is the dimension of the hidden vector. The outputs of related proposition modules $p_1, ...p_M$ in the previous layer are vectors $\psi_{p_1}^{l-1}, ..., \psi_{p_M}^{l-1}$. These vectors, together with the output vector of the same action module from the previous layer ϕ_a^{l-1} are concatenated to form the input vector u_a^l, which is then fed into an affine transform $f(W_a^l \cdot u_a^l + b_a^l) = \phi_a^l$. All actions a from the same action schema i in this layer share the same weights and bias, namely $W_a^l = W_i^l, b_a^l = b_i^l$.

Similarly, a proposition module at layer l with underlying proposition p takes in a vector $v_p^l \in \mathbb{R}^{d_p^l}$ and outputs another vector $\psi_p^l \in \mathbb{R}^{d_h}$. Instead of concatenation, the outputs of related action modules $\phi_{a_1}^{l-1}, ..., \phi_{a_N}^{l-1}$ from the previous layer are grouped by their position of relatedness and then passed through a max pooling function. Then the outputs of the pooling functions are concatenated with the output vector of the same proposition module from the previous layer ψ_p^{l-1}. Finally, the resulting vector v_p^l is fed into an affine transform $f(W_p^l \cdot v_p^l + b_p^l) = \psi_a^l$. The weights W_i^l and bias b_i^l are shared by all propositions grounded from the same predicate in this layer.

Because weights are shared between actions grounded from the same action schema (same for propositions of the same predicate), the learned weights are generic transformations that can be applied to the instantiated networks for of problems of any size.

The input features to the first action layer are encodings of the current states. For each action module with related propositions $p_1, ..., p_M$, the input vector u_a^1 consists of four parts: a vector $v \in \{0, 1\}^M$ where $v_i = 1$ iff the corresponding proposition p_i is true in current state; a vector

$g \in \{0,1\}^M$ where $g_i = 1$ iff the proposition appears unnegated in the goal; a scalar $m = 1$ if the action is applicable in current state; and finally, an extension part including features derived from other heuristics. The first half of the extension part is a one-hot label vector $c \in \{0,1\}^3$ derived from LM-cut landmarks (Helmert and Domshlak 2009). The vector has $c_1 = 1$ iff the action appears as the only action in at least one landmark; $c_2 = 1$ iff the action appears in a landmark containing two or more actions; and $c_3 = 1$ otherwise. The second half of the extension is an integer recording the number of times the action has appeared in the plan previously.

The last layer is also an action layer but with different outputs. The output dimension of each action module is one. Specifically, each action module outputs a single scalar value $\phi_{a_1}^{L+1}, ..., \phi_{a_N}^{L+1}$. The final policy is then the log-scaled probability distribution:

$$\pi^\theta(a_i|s) = \frac{m_i \text{exp}(\phi_{a_i}^{L+1})}{\sum_{j=1}^{N} m_j \text{exp}(\phi_{a_j}^{L+1})}.$$

The model is then trained through imitation learning. Namely, the model will try to imitate the action selection policy of a teacher planner. At each training step, ASNets sample a set of states (so-called "rollouts") and try to find a plan from the states to the goal by choosing actions according to the learnt policy π^θ. Then the trainer compares these plans to solutions found by the teacher planner and learns from their differences. Originally ASNets used SSiPP (Trevizan and Veloso 2014)) for stochastic problems and fast-downward (Helmert 2006) for deterministic problems. The teacher outputs a Q-value $Q^{teach}(s, a)$, which represents the cost of reaching the goal from state s when taking action a and acting optimally afterwards. Therefore, the best action will have the lowest Q-value. Using the Q-values, the trainer calculates the cross-entropy loss between the learnt policy π^θ and the teacher's policy. By minimising the loss, the model learns to make similar decisions as the teacher planner.

2 Changes to ASNets

In this section we describe our changes and improvements on the original implementation of ASNets.

2.1 ML Framework

The learning module of the original ASNets is based on TensorFlow 1.x (TF1) (Abadi 2016), which is a deprecated ML framework with very different runtime behaviour compared to the latest frameworks. We updated the implementation by replacing the learning framework with TensorFlow 2.x (TF2) (Singh et al. 2020), which has faster and simpler API and more supported functions. Some major differences between TF1 and TF2 includes:

- TF1 uses lazy execution and requires manual graph construction representing the model architecture (Abadi 2016), and then manually compiles the graph by passing the input and output tensors to a $Session$. TF2 executes eagerly (Singh et al. 2020). The construction of the graph can be done in the background by calling higher-level API modules.
- TF1 variables are based on implicit global namespaces. Model variables are stored in the graph even if no Python variable is pointing to it. TF2 variables are managed the same way as Python variables, making it faster and more memory-efficient.
- TF2 provides a well-defined high-level API called Keras (Gulli and Pal 2017) to easily build and train a neural network while remaining highly customisable.

We re-implemented the model and model trainer using the Keras API. The behaviours of the action and proposition modules are defined in separate classes. Connections between the modules are defined in a network extended from the Keras Layer class, which provides a well-defined API for training and evaluation. The shared weights are managed through a weight manager where weights are defined as TensorFlow variables with learnable values. Then we used a Keras Loss Class to implement the loss function described in (Toyer et al. 2020) to use the class interface in training directly. Finally, we rewrote the trainer with the new models and components while keeping the data generation and training mechanism unchanged.

2.2 Training

Originally ASNets were designed for solving Stochastic Shortest Path problems (SSP) (Bertsekas and Tsitsiklis 1996). Because we only consider deterministic domains in this competition, we simplified and optimised the model for the deterministic setting.

First, for the teacher planner, we followed the configuration of the deterministic version of the original ASNets, which used fast-downward (Helmert 2006) in solving deterministic problems. In other words, we use fast-downward to compute the teacher's policy $Q^{teach}(s, a)$ for state s and action a during the training process. And for the solver algorithm used by fast-downward, we chose the LAMA-first algorithm (Richter and Westphal 2010), whereas the original version of ASNets used A-star with LM-cut (Helmert and Domshlak 2011). In stochastic problems, the policy is generally represented by a probability distribution function, while in deterministic settings, we simply take the best action. Therefore, the Q-value computed by the teacher becomes a one-hot label indicating which action is the optimal selection and rejecting all other actions.

Because ASNets can be trained on small problems and then reuse the same learnt weights for solving larger problems, it is unnecessary to train the model on large problems directly. However, in this competition, we don't know the domains and sizes of problems in advance. Hence we cannot set the boundary between "easy" and "hard" problems[1]. Because generating training data for the model requires solving many sub-problems (namely, finding plans from some intermediate states to the goal), if a training problem is too hard

[1]The competition provides problems in increasing order of difficulty

to solve, the learning script might waste too much time solving these sub-problems instead of training the model. Therefore, the program has to decide by itself whether to use the given training problem or not.

We set the timeout threshold of the fast-downward solver to 1 minute. In other words, if the teacher fails to find a plan for the given state within one minute, it will skip the problem and train the model with data from other problems. Besides, we train the model in an incremental manner: we first train the model with only the two easiest problems. The learnt policy is used as a basic planner. Then we append one more problem to the training set and train a new model. We repeat this process until all problems are in the training set or the teacher fails to return plans for all new problems. We record the problems that failed to be solved by the planner and then try to train the model again with fewer rollouts per step and a longer time limit (2 minutes) for the teacher planner. The incremental training set strategy guarantees we can have a benchmark model before the trainer runs out of memory or fails to learn from harder problems.

We used a stepped decreasing learning rate starting from $1e^{-3}$. The learning rate drops to $1e^{-4}$ after 500 steps and further drops to $1e^{-5}$ after 1000 steps. The L1 regularisation parameter is set to 0, and the L2 regularisation parameter is set to $2e^{-4}$. The model has 3 action layers and 2 proposition layers with hidden vector size 16 and dropout rate 0.1. We save the trained model after every epoch if the evaluation result is better than the previous one.

2.3 Evaluation

The original ASNets select an action to execute in a state s by identifying the action with the highest probability in its last layer, i.e., $\arg\max_{a \in A(s)} \pi^{\theta}(a|s)$, breaking ties arbitrarily. While effective in practice, ASNet does not guarantee that the generated plans will be loop-free. For example, in a Blocks World problem, the $\arg\max$ strategy cannot guarantee that picking up block A from the table, placing A back on the table, and repeating the cycle indefinitely will never happen. The solution proposed in (Toyer et al. 2020) is to sample the action to be executed in s according to $\pi^{\theta}(\cdot|s)$, guaranteeing that loops will be eventually broken; however, this method performs worse than the $\arg\max$ strategy in practice because the probability of sampling an unfavorable action is higher than the probability of the $\arg\max$ strategy resulting in a loop. To address these issues, we use a new execution method based on the $\arg\max$ strategy that ensures loop-free execution while preserving its performance in the absence of loops. To do so, the planner keeps track the actions executed in each visited state s, denoted as $E(s)$. An action for state s is then chosen according to $\arg\max_{a \in A(s) \setminus E(s)} \pi^{\theta}(a|s)$, i.e., we select the highest probability action that has not yet been executed. If there are no applicable actions remaining, i.e., $E(s)$ equals $A(s)$, the planner returns a failure to find a plan. This strategy behaves as the $\arg\max$ strategy until a loop is detected, in which case it systematically executes different actions from highest to lowest probability according to $\pi^{\theta}(\cdot|s)$.

References

Abadi, M. 2016. TensorFlow: learning functions at scale. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 1–1.

Bertsekas, D.; and Tsitsiklis, J. N. 1996. *Neuro-dynamic programming*. Athena Scientific.

Gulli, A.; and Pal, S. 2017. *Deep learning with Keras*. Packt Publishing Ltd.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, critical paths and abstractions: what's the difference anyway? In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 19, 162–169.

Helmert, M.; and Domshlak, C. 2011. Lm-cut: Optimal planning with the landmark-cut heuristic. *Seventh international planning competition (IPC 2011), deterministic part*, 103–105.

Richter, S.; and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39: 127–177.

Singh, P.; Manure, A.; Singh, P.; and Manure, A. 2020. Introduction to tensorflow 2.0. *Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python*, 1–24.

Toyer, S.; Thiébaux, S.; Trevizan, F.; and Xie, L. 2020. Asnets: Deep learning for generalised planning. *Journal of Artificial Intelligence Research*, 68: 1–68.

Toyer, S.; Trevizan, F.; Thiébaux, S.; and Xie, L. 2018. Action schema networks: Generalised policies with deep learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

Trevizan, F. W.; and Veloso, M. M. 2014. Depth-based short-sighted stochastic shortest path problems. *Artificial Intelligence*, 216: 179–205.