FSM — A Short-Time Learning Planner

Gustavo Prolla Lacroix, Rafael Vales Bettker, André Grahl Pereira

Federal University of Rio Grande do Sul, Brazil
{gplacroix, rvbettker, agpereira}@inf.ufrgs.br

Abstract

FSM is a learning-based planner that uses different sample generation improvement strategies to learn a heuristic function for classical planning. The heuristic function is a neural network trained on pairs of states and cost-to-goal estimates generated by regression from the goal. Samples are generated from breadth-first search, random walk, and random sampling algorithms and post-processed to improve the cost-to-goal estimates. The short time for sampling and training allows FSM to compete as a satisficing planner.

Introduction

FSM brings a learning-based model to the classic track of the International Planning Competition (IPC) 2023. Learning heuristic functions using neural networks (NN) (Ferber et al. 2022; O'Toole et al. 2022; Yu, Kuroiwa, and Fukunaga 2020) is a way to generate model-free heuristic functions with competitive results when compared to logic-based heuristics such as h^{FF}. In particular, the FSM planner is based on the sampling techniques proposed by Bettker et al. (2022) implemented in the Neural Fast Downward (Ferber, Helmert, and Hoffmann 2020b).

The planner works in three stages: sampling, training, and searching. Sampling generates pairs of states and cost-to-goal estimates used to learn a heuristic function. FSM uses different sampling algorithms to achieve a good distribution of states across the state space. As post-processing, two techniques for improving the cost-to-goal estimates are applied, sample improvement and successor improvement. (Sampling is discussed in detail in the next section.) Next, the samples feed a feedforward neural network to learn a state space-specific heuristic function. The NN takes the state in the STRIPS representation as input and generates the h-value that guides the greedy best-first search (GBFS) to solve the task.

We allocate the 30-minute search time as follows: 15 minutes for sampling, 10 minutes for training, and 5 minutes for the actual search. The timers between the stages are shared, i.e., if a stage finishes before the timeout, its remaining time is added to the next stage, so the real-time allocation varies according to the task.

Sampling Stage

In this section, we describe four sample generation strategies: two that aim to generate more representative states and two that enhance the cost-to-goal estimates.

Sample Set

Samples are generated by regression from the goal state. FSM planner generates $N = 25000$ samples (or until timed out), where 10% breadth-first search (BFS), 70% random walk (RW), and 20% random sampling. First, the BFS followed by RW, where the random walk paths (rollouts) start from the BFS leaf nodes until a regression limit L is reached. Then, we randomly sample uniformly over the state space as proposed by O'Toole et al. (2022). States generated by the regression are given the rollout depth as cost-to-goal estimate h, while random samples receive $1+\max_{i\in\{1,\dots,N\}} h_i$. The regression generates samples as partial states, whose undefined values are randomly assigned (respecting the mutexes) at the end of the sampling stage to train the NN with complete states.

Regression Limit

Since the cost-to-goal estimate of the samples is based on the distance traveled from the goal state to the sampled state, the chances (and magnitude) of the cost-to-goal estimate being overestimated increase with the length of the rollout. Limiting the rollout – and consequently, the maximum h-value of a sample – with a fixed number is not a good option since we do not have prior information about the task's state space. A short rollout may prevent sampling far-from-goal states, while a long rollout can overestimate the cost-to-goal estimates.

Therefore, we use the task-based strategy of limiting the random walk length by the number of propositions divided by the average number of effects on operators, i.e., $L = \lceil |\mathcal{P}| / \overline{\text{eff}} \rceil$ where $\overline{\text{eff}} = \sum_{o\in\mathcal{O}} |\,\text{eff}(o)|/|\mathcal{O}|$ for propositions set $\mathcal{P}$ and operators set $\mathcal{O}$. This technique generates an approximate estimate of the diameter of the state space (largest h^* of the state space) considering that, in the worst case, we need to set all propositions of a state to reach the goal state, and we can set on average $\overline{\text{eff}}$ propositions per given step, so we need to take L steps to traverse the state space.

Sample Improvement

Generating samples from multiple rollouts allows an intersection between them, where two samples of the same state can have different cost-to-goal estimates. Training the NN with different labels for the same sample can lead to inconsistency, so we apply sample improvement (SAI). By definition, the cost-to-goal estimate is never underestimated when set to the rollout depth so that the minimum value will be the closest to h^*. Thus, SAI is a post-processing technique to assign equal samples the minimum cost-to-goal estimate found between them, i.e., $h(s) = \min\{h_i \mid s = s_i, i \in \{1, \ldots, N\}\}$. Since different partial states can generate the same complete state, the technique is applied both while the samples are in partial state and complete state.

Successor Improvement

The cost-to-goal estimate of sampling by regression is based solely on the rollout it was generated in, allowing different rollouts to sample two close states but with significantly different cost-to-goal estimates. While SAI improves different samples from the same state, successor improvement (SUI) improves based on neighboring states in state space, as follows. Consider a directed graph $G = (V, A)$ over all sampled partial states, i.e., $V = \{s_i \mid i \in \{1, \ldots, N\}\}$. For every pair of states $s, t \in V$ such that for some operator $o \in \mathcal{O}$ applicable to s we have $succ(s, o) \subseteq t$, we add an arc (s, t) of length $cost(o)$ to A. For fast subset, we keep all samples in a trie and search for each successor in the states that are supersets. For partial states generated by regression, by construction, at least one such successor exists, except for the goal state. We then compute the shortest paths to the goal in graph G via the Dijkstra algorithm and update the cost-to-goal estimates with these distances.

Learning Stage

The sample set feeds a residual neural network (He et al. 2016) to learn a state space-specific heuristic function. The structure of the NN is proposed by Ferber, Helmert, and Hoffmann (2020a), which has two hidden layers followed by a residual block with two hidden layers. The input layer has a number of neurons equal to the total propositions of the task, while the hidden layers contain 250 neurons with ReLU activation and He initialization (He et al. 2015). The output is a single neuron with the predicted h-value. We use the Adam optimizer (Kingma and Ba 2015), a learning rate of 10^{-4}, a mean squared error loss function, and a batch size of 512. The data is split into 90% for the training set and the remainder for the validation set. The NN receives the states in a boolean representation where, for each proposition of the task, the value 1 is given if the proposition is true in the state, otherwise 0. Learning terminates by early stopping after 25 epochs without improvement (as opposed to 100 proposed by Bettker et al. (2022) since time is valuable) or when reaching the time limit, and the best epoch model is the heuristic function in the search.

Search

We use Fast Downward (Helmert 2006) to solve tasks with GBFS guided by the learned heuristic. All arguments remain with their default values, except for considering unit costs for all tasks.

Acknowledgments

We thank the Fast Downward planning system team. We especially thank Patrick Ferber, Malte Helmert, and Jörg Hoffmann for sharing the source code of the Neural Fast Downward planning system. This study was financed in part by the *Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil* (CNPQ) - PIBIC.

References

Bettker, R. V.; Minini, P. P.; Pereira, A. G.; and Ritt, M. 2022. Understanding Sample Generation Strategies for Learning Heuristic Functions in Classical Planning. arXiv:2211.13316.

Ferber, P.; Geißer, F.; Trevizan, F.; Helmert, M.; and Hoffmann, J. 2022. Neural Network Heuristic Functions for Classical Planning: Bootstrapping and Comparison to Other Methods. In *International Conference on Automated Planning and Scheduling*.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020a. ICAPS Reinforcement Learning for Planning Heuristics. In *ICAPS Workshop on Bridging the Gap Between AI Planning and Reinforcement Learning*, 119–126.

Ferber, P.; Helmert, M.; and Hoffmann, J. 2020b. Neural Network Heuristics for Classical Planning: A Study of Hyperparameter Space. In *European Conference on Artificial Intelligence*, 2346–2353.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *International Conference on Computer Vision*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26: 191–246.

Kingma, D.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations*.

O'Toole, S.; Ramirez, M.; Lipovetzky, N.; and Pearce, A. R. 2022. Sampling from Pre-Images to Learn Heuristic Functions for Classical Planning. In *Symposium on Combinatorial Search*, volume 15, 308–310.

Yu, L.; Kuroiwa, R.; and Fukunaga, A. 2020. Learning Search-Space Specific Heuristics Using Neural Network. In *ICAPS Workshop on Heuristics and Search for Domain-independent Planning*.