Reinforcement Learning with A* and a Deep Heuristic

Ariel Kesleman
Imagry
ariel@imagry.co

Sergey Ten
Imagry
sergey@imagry.co

Adham Ghazali
Imagry
adham@imagry.co

Majed Jubeh
Imagry
majed@imagry.co

Abstract

A^* is a popular path-finding algorithm, but it can only be applied to those domains where a good heuristic function is known. Inspired by recent methods combining Deep Neural Networks (DNNs) and trees, this study demonstrates how to train a heuristic represented by a DNN and combine it with A^* . This new algorithm which we call $\aleph^*$ can be used efficiently in domains where the input to the heuristic could be processed by a neural network. We compare $\aleph^*$ to N-Step Deep Q-Learning (DQN Mnih et al. 2013) in a driving simulation with pixel-based input, and demonstrate significantly better performance in this scenario.

1 Introduction

The problem of finding the minimal cost path between two nodes on a graph can be formulated as a decision process where at every visited node an optimal action has to be taken minimizing the total accumulated cost. Replacing cost by reward, any such algorithm that generates cost minimizing actions generates reward maximizing actions thus becoming a candidate solver for Markov Decision Processes (MDPs). A^* (Hart et al., 1968) is such a cost minimization algorithm that takes advantage of domain knowledge in the form of a heuristic function; it is interesting because for certain conditions (admissibility and consistency of the heuristic) A^* converges to the optimal solution while visiting a minimal number of nodes, i.e. no other similar algorithm (equally informed) could perform better.

The strength of A^* is also its main weakness: when a heuristic is unknown other algorithms are used, most notably Monte Carlo Tree Search (MCTS, Abramson 1987). MCTS randomly samples the configuration space and balances exploration vs exploitation. Most modern implementations are based on the Upper Confidence bound applied to Trees (UCT, Kocsis and Szepesvári 2006). MCTS is a rollout based method, nodes are searched by walking the tree from the root, leafs are evaluated by performing (possibly) random actions until the episode ends. MCTS was successfully used for games like go since the early 90's (Brügmann, 1993; Gelly et al., 2006), and more recently with greater success by replacing the random rollout with a deep convolutional network (Silver et al., 2016; Anthony et al., 2017; Silver et al., 2017b,a).

Following the recent work with MCTS, in this study we address the main weakness of A^* by replacing the rule-based heuristic with a Neural Network, specifically by a Convolutional Neural Network (CNN), resulting in the algorithm $\aleph^*$. The weights are learned in a Reinforcement Learning (RL) fashion, interacting with a simulated environment by performing actions and earning rewards. The algorithm is model-based, episodes are not linear, all visited states and their possible actions are kept in a priority queue for later consideration. Transition of states are kept in a tree structure[1]. Action values Q are backpropagated along the tree to satisfy a variant of time-difference equation. Unlike a common time-difference equation which use maximum of Q over actions we choose to use a variant

[1] ideally should be a directed graph since different actions could lead to the same state, but it is not obvious how much this would help in practice

of "soft" maximum:

$$Q_a^s = r_a^s + \gamma \frac{\sum_{b \in \mathcal{A}} Q_b^x w_b^x}{\sum_{b \in \mathcal{A}} w_b^x} \ . \tag{1}$$

Here γ is the discount factor, r_a^s denotes the reward given by performing action a on state s, $\mathcal{A}$ denotes the set of possible actions which without loss of generality is assumed to be similar across all possible states, x denotes the state resulting from taking action a on state s, and w is an action value weighting factor as explained in sec. 2. The heuristic $\mathcal{H}_\theta$ is represented by a deep neural network with parameters θ and taking as input the sensors S, themselfs a function of state[2]. The heuristic is trained to predict the action values by using some loss function, in the reference implementation we use

$$\mathcal{L} = \frac{|Q^s - \mathcal{H}_\theta(S(s))|^2}{\text{len}(\mathcal{A})} \tag{2}$$

where Q^s is a vector of action values for state s with an entry for every possible action $a \in \mathcal{A}$.

In our experiments the resulting tree is very efficient, it has little branching, a property enabling its use in runtime. The reason for this is that while $\aleph^*$ uses random exploration during training, it uses pure exploitation during evaluation. Whether this is better or worse than UCT depends of course on the quality of the heuristic, and remains an open question.

The structure of this paper is as follows: in section 2 we describe in detail the $\aleph^*$ algorithm and a reference implementation, in section 3 we describe the experiment and environment used in this study, in section 4 we present the main results together with an explanation on reproduction, and we conclude with a summary in section 5.

2 The Algorithm

$\aleph^*$ is a general model-based reinforcement learning algorithm and it uses the regular notion of an agent interacting with a simulated environment. A reference implementation of the algorithm described here can be found at `https://github.com/imagry/aleph_star`.

The training algorithm consists of building randomly initialized trees as described in Alg. 1. The basic building block of the tree is a node, which contains the necessary information to maintain the tree structure (pointers to its parent and children), and also important information to perform the backpropagation of the Q values from leafs to root according to the weighted time-difference as required by equation 1. Possible actions and their parent nodes are added to a priority queue maximizing $C + Q_a$ where C is the (non-discounted) total accumulated reward. Actions chosen for expansion not only maximize future discounted reward like in DQN, but they take into consideration also actual reward. After a tree is backpropagated, the experiences are stored in an experience buffer, which is then used to update the heuristic weights θ using any gradient descend method. The experience buffer is implemented as an array of tuples $[(Q^s, S(s)), \ldots]$ each containing a vector of action values and the sensors for the same state. It is built by iterating over tree nodes, calculating the sensors, and pushing new tuples to the array.

The weights w_b^x in eq. 1 are the number of sub-nodes explored following action b. This results in larger weights for more thoroughly explored actions. For a good heuristic most propagation would happen through a single main tree "trunk" as it would dominate weights. In the extreme this just converges to N-Step DQN. For completely unexplored nodes we set $w = 1$ for all actions. Other weighting schemes could be used, for eg.g setting $w = 0$ and $w_{argmax(Q)} = 1$ which propagates through action-value maximizing branches (classical Q-Learning), a soft-max strategy, or weights that depend on the depth of the nodes. Also a threshold to w could be applied, but we find that our simple strategy for weighting works good enough. Backpropagating according to Eq. 1 is easy to implement if child nodes are linearly added to the tree array always after their parent: iterating in reverse guarantees that all children of the current node were previously visited.

Finally, after each training iteration the exploration parameter ϵ and the learning rate can be updated just like in DQN. Many of the improvements applied to DQN can be implemented for $\aleph^*$ too: the experience replay can include a priority based on loss Schaul et al. (2015) (as implemented in accompanaing code), exploration rewards based on curiosity can be added Burda et al. (2018),

[2] this formalism allows accommodating different sensors per state

rewards can be clipped Pohlen et al. (2018), a target network can be used, and so on. In the reference implementation instead of using a target network we remove nodes that are too old from the experience buffer. This has the advantage of evaluating the heuristic only once per experience. Another detail regards the implementation of the priority queue: for efficient insertion and removal of values it is implemented using a heap. Efficiently popping random values from a heap is hard to do, hence instead of deleting values we just tag them as used. When the ratio of number of used to unused values crosses a threshold a garbage-collection cycle is run, it rebuilds the data-structure without the tagged entries.

For the runtime algorithm the heuristic can be used by itself just by calculating the sensorsy input, evaluating the heuristic and performing the action that maximizes the action value. Another option is to create a small tree guided by the heuristic with $\epsilon = 0$, and acting on the action that leads from the root node to the last added node. When $\epsilon = 0$ the last added node maximizes $Q_a + node.C$ i.e. accumulated reword plus "past" reward. This is different than DQN or MCTS which just maximize Q. If the heuristic is good, the tree can be much thinner than an MCTS, allowing even small trees to be highly effective.

Note that Alg. 1 assumes infinite long episodes i.e. ending an episode is always considered a bad thing getting the minimal value possible of 0. This requires always positive rewards, and positive heuristic. This heuristic behaviour can be enforce in numerous ways, for e.g. by passing the output through an $\mathrm{abs}()$ non-linearity at the end of the neural network. Alg. 1 also assumes a minimum of two nodes per tree, as it is arguable whether a smaller tree can still be called a tree.

3 Experiment

We implemented a pixel-based driving environment, the sensors are represented as an 84x84 grayscale image, the heuristic architecture is mostly similar to the one used by Mnih et al. (2013), it consists of a convolution layer of 16 8x8 filters with stride 4 followed by a non-learnable layer normalization (Ba et al., 2016) and a leaky-relu non-linearity (Xu et al., 2015) with $\alpha = 0.3$. The second convolutional layer consists of 32 4x4 filters with stride 2 similarly followed by layer normalization and leaky-relu. The last hidden layer is fully connected with 256 linear units, a layer normalization, and leaky-relus. The output layer consists of 35 linear units, as the number of actions in $\mathcal{A}$ (7 steering angles and 5 accelerations).

The sensors can be seen in Fig. 1. Instead of using multiple frames as input, and since the simulation was developed by us, we chose to encode temporal information (the information affecting system dynamics such as velocity and steering angle) in the values of pixels: car velocity relative to the road is encoded in the background and in the color of the car itself. Relative velocity to other cars is encoded in the color of the target cars; and finally steering angle is encoded in the color of the vertical edges of the image.

The reward used is proportional to the car velocity, and multiplied by a factor preferring central positions and angles tangential to the lane. A small constant "keep-alive" reward is also added every simulated time-step. The heuristic was trained with SGD, learning rate kept constant 0.01, and gamma 0.98. Batch size 64, and ϵ changed from an initial value of 0.5 to 0.01. Maximum tree size was 5500 nodes, and training continued for 1000 iterations. The N-Step DQN implementation closely follows $\aleph^*$, except there is no priority queue, and branching is not allowed. There are plenty more details, all are present in the reference implementation.

4 Results

The results are summarized in Fig. 2. Looking at the left panel. Training of the heuristic using $\aleph^*$ was efficient at 1000 iterations, reaching up to 50% of the theoretical upper bound cumulative reward (given maximum number of nodes). Using the heuristic alone without a tree resulted in 50% performance reduction (down to 25% of the theoretical maximum) and N-Step DQN was not able to learn effectively. We also tried a few runs of N-Step DQN starting with $\epsilon = 1$ but results were similar.

We define the rank of an $\aleph^*$ tree as the depth of the node maximizing accumulated reward C + max(Q). The efficiency is defined as the ratio of rank to maximal tree size. The efficiency is plotted in the right panel of Fig. 2: the $\aleph^*$ tree, when trained, has little branching, and almost 85% of the

Algorithm 1 Generate a tree guided by the heuristic $\mathcal{H}_\theta$

initialize an empty vector of nodes $Tree$
initialize a priority queue $Queue$
initialize the root node:
$node \leftarrow$ new empty node
$node.s \leftarrow$ randomly initialized state ▷ save the initial state in the root node
$node.C \leftarrow 0$ ▷ accumulated (past) reword is zero for root
$node.R \leftarrow 0$ ▷ reward given by performing action leading to this root node
$node.Done \leftarrow$ False ▷ root is a non-terminal state
$node.Q \leftarrow 0$) ▷ vector of action values, one entry per action
$node.Parent \leftarrow$ None ▷ root node has no parent
$node.Children \leftarrow \{\}$ ▷ a dictionary actions $\rightarrow$ children nodes
$Tree \leftarrow node$ ▷ append root node to tree
repeatedly add nodes until tree is large enough:
repeat
    $node.Q \leftarrow \mathcal{H}_\theta(S(s))$
    if not $node.Done$ **then**
        for all $a \in \mathcal{A}$ **do**
            $Queue \leftarrow (a, node)$ with priority $node.C + node.Q_a$
        end for
    end if
    if $rnd() < \epsilon$ **then** ▷ exploration parameter
        $a, node \leftarrow Queue.popRand()$
    else
        $a, node \leftarrow Queue.popMax()$
    end if
    $newState, reward, done \leftarrow simulate(node.s, a)$
    $newNode \leftarrow$ new empty node
    $newNode.s \leftarrow newState$
    $newNode.C \leftarrow node.C + reward$ ▷ no discount for past accumulated reward
    $newNode.R \leftarrow reward$
    $newNode.Done \leftarrow done$
    $newNode.Q \leftarrow 0$
    $newNode.Parent \leftarrow node$
    $node.children[a] \leftarrow newNode$
    $Tree \leftarrow newNode$ ▷ append node to tree
    $node \leftarrow newNode$
until len($Queue$) == 0 or len($Tree$) >= maximum number of nodes in tree
return $Tree$

nodes are sequential. This means that a tree of only 10 nodes could, on average, be used for planning to a depth of 8 time-steps. Ity could be feasible to run such a tree on runtime. In the experiment trees regularly grow beyond a rank of 4000 while training. Since node choice is being done by a heap powered priority queue, there is no need to walk all these nodes in order to choose the next node to explore. Instead pushing or popping a single node from the queue takes constant time $\mathcal{O}(1)$ (for e.g. using a Fibonacci heap see Fredman and Tarjan 1987), and hence building such tree scales as $\mathcal{O}(rank)$. In contrast, rollout based algorithms like MCTS scale as $\mathcal{O}(rank^2)$, potentially becoming a bottleneck when reaching rank in the thousands.

5 Summary

We presented a new model based reinforcement learning algorithm that efficiently combines a tree and a learnable heuristic. This algorithm has roots in A^* which was proven to be optimal under certain conditions. We coded a reference implementation and open-sourced it at `https://github.com/imagry/aleph_star`. We tested the reference implementation in a pixel based sensory context and compared it to N-Step DQN, and found that while N-Step DQN fails to learn, $\aleph^*$ learns very effectively. Furthermore, building of the tree is efficient even at depths of thousands,

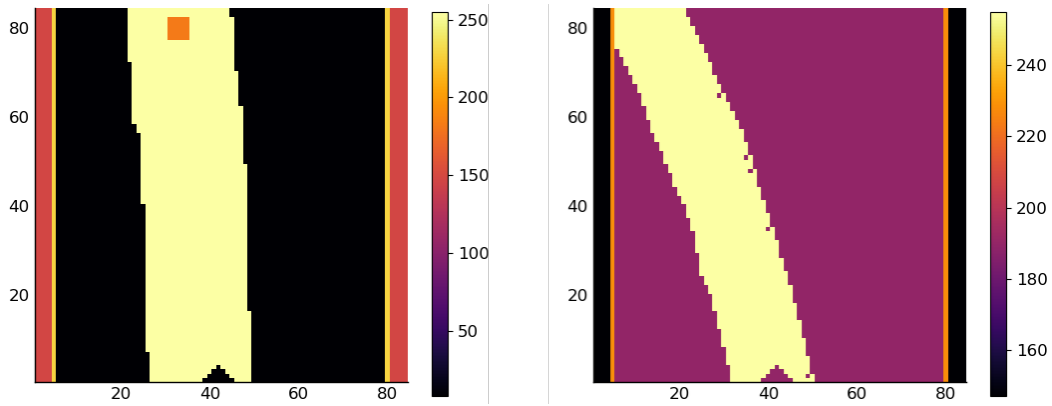

Figure 1: Two sampled sensors from different environment initialization. The sensors include a road, the actor car, other cars, and some information encoded in the pixel values as described in the text. The sensors are grayscale, a color palette is used here for better presentation.

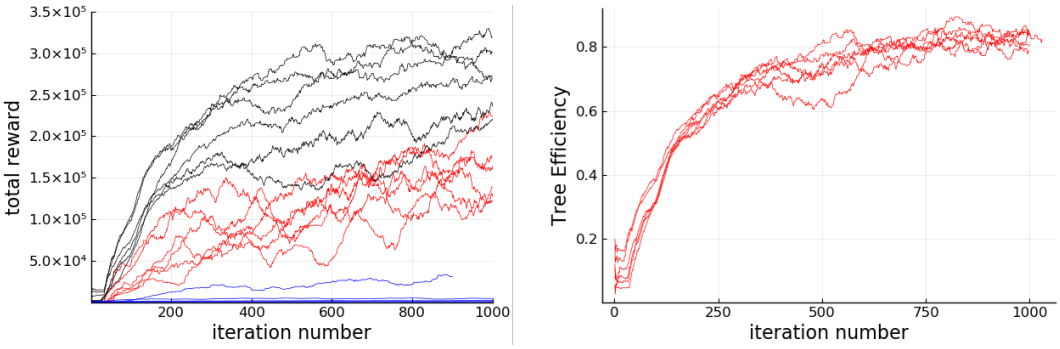

Figure 2: Main results for a few randomly initialized environments. Left panel accumultated reward as function of training iterations with a moving average window of 50 (black for the 5500 node tree, red for the heuristic alone, and blue for N-Step DQN), right panel tree efficiency as defined in the text

which could otherwise become a bottleneck in algorithms with rollouts such as MCTS. There is still much work to do, for e.g. a proper comparison to MCTS or AlphaZero, testing new environments, etc.

Acknowledgments

References

Abramson, B. D. (1987). The expected-outcome model of two-player games.

Anthony, T., Tian, Z., and Barber, D. (2017). Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370.

Ba, J. L., Kiros, J. R., and Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Brügmann, B. (1993). Monte carlo go. Technical report, Citeseer.

Burda, Y., Edwards, H., Storkey, A., and Klimov, O. (2018). Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*.

Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615.

Gelly, S., Wang, Y., Teytaud, O., Patterns, M. U., and Tao, P. (2006). Modification of uct with patterns in monte-carlo go.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107.

Kocsis, L. and Szepesvári, C. (2006). Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Pohlen, T., Piot, B., Hester, T., Azar, M. G., Horgan, D., Budden, D., Barth-Maron, G., van Hasselt, H., Quan, J., Večerík, M., et al. (2018). Observe and look further: Achieving consistent performance on atari. *arXiv preprint arXiv:1805.11593*.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. (2016). Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2017a). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017b). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

Xu, B., Wang, N., Chen, T., and Li, M. (2015). Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*.