TREEQN AND ATREEC: DIFFERENTIABLE TREE PLANNING FOR DEEP REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Combining deep model-free reinforcement learning with on-line planning is a promising approach to building on the successes of deep RL. On-line planning with look-ahead trees has proven successful in environments where transition models are known a priori. However, in complex environments where transition models need to be learned from data, the deficiencies of learned models have limited their utility for planning. To address these challenges, we propose TreeQN, a differentiable, recursive, tree-structured model that serves as a drop-in replacement for any value function network in deep RL with discrete actions. TreeQN dynamically constructs a tree by recursively applying a transition model in a learned abstract state space and then aggregating predicted rewards and state-values using a tree backup to estimate Q-values. We also propose ATreeC, an actor-critic variant that augments TreeQN with a softmax layer to form a stochastic policy network. Both approaches are trained end-to-end, such that the learned model is optimised for its actual use in the planner. We show that TreeQN and ATreeC outperform n-step DQN and A2C on a box-pushing task, as well as n-step DQN and *value prediction networks* (Oh et al., 2017) on multiple Atari games, with deeper trees often outperforming shallower ones. We also present a qualitative analysis that sheds light on the trees learned by TreeQN.

1 INTRODUCTION

A promising approach to improving model-free deep reinforcement learning (RL) is to combine it with on-line planning. The model-free value function can be viewed as a rough global estimate which is then locally refined on the fly for the current state by the on-line planner. Crucially, this does not require new samples but only additional computation, which is often available.

One strategy for on-line planning is to use look-ahead tree search (Knuth & Moore, 1975; Browne et al., 2012). Traditionally, such methods have been limited to domains where perfect environment simulators are available, such as board or card games (Coulom, 2006; Sturtevant, 2008). However, in general, models for complex environments with high-dimensional observation spaces and complex dynamics must be learned from agent experience. Unfortunately, to date, it has proven difficult to learn models for such domains with sufficient fidelity to realise the benefits of look-ahead planning (Oh et al., 2015; Talvitie, 2017).

A simple approach to learning environment models is to maximise a similarity metric between model predictions and ground truth in the observation space. This approach has been applied with some success in cases where model fidelity is less important, *e.g.*, for improving exploration (Chiappa et al., 2017; Oh et al., 2015). However, this objective causes significant model capacity to be devoted to predicting irrelevant aspects of the environment dynamics, such as noisy backgrounds, at the expense of value-critical features that may occupy only a small part of the observation space (Pathak et al., 2017). Consequently, current state-of-the-art models still accumulate errors too rapidly to be used for look-ahead planning in complex environments.

Another strategy is to train a model such that, when it is used to predict a value function, the error in those predictions is minimised. Doing so can encourage the model to focus on relevant features of the observations. An example is the *predictron* (Silver et al., 2017b), where the model is used to aid policy evaluation without addressing control. *Value prediction networks* (VPNs) (Oh et al., 2017) take a similar approach but use the model to construct a look-ahead tree only when constructing

bootstrap targets and selecting actions, similarly to *TD-search* (Silver et al., 2012). Crucially, the model is not embedded in a planning algorithm during optimisation.

We propose a new approach to address the aforementioned problems: by formulating the tree search in a differentiable way and integrating it directly into the Q-function or policy, we train the entire agent, including its learned model, end-to-end, thus ensuring that the model is optimised for the correct goal and is suitable for on-line planning during execution of the policy.

Our approach can alternatively be viewed as a model-free method in which the fully connected layers of DQN are replaced by a recursive network that applies transition functions with shared parameters at each tree node expansion. The resulting architecture, which we call TreeQN, encodes an inductive bias based on the prior knowledge that the environment is a stationary Markov process, which facilitates faster learning of better policies. We also present an actor-critic variant, ATreeC, in which the tree planner is augmented with a softmax layer and used as a policy network.

The main contributions of this paper are: (i) we introduce TreeQN and ATreeC, end-to-end differentiable tree planners that serve as drop-in replacements of value and policy networks in deep RL with discrete actions; (ii) we show that TreeQN and ATreeC outperform their DQN-based counterparts in a box-pushing domain and a suite of Atari games, with deeper trees often outperforming shallower trees, and that TreeQN outperforms VPN (Oh et al., 2017) on most Atari games; (iii) we present a qualitative analysis of the learned trees in the box-pushing domain.

2 BACKGROUND

We consider an agent learning to act in a Markov Decision Process (MDP), with the goal of maximising its expected discounted sum of rewards $R_t = \sum_{t=0}^{\infty} \gamma^t r_t$, by learning a policy $\pi(\mathbf{s})$ that maps states $\mathbf{s} \in \mathcal{S}$ to actions $a \in \mathcal{A}$. The state-action value function (Q-function) is defined as $Q^\pi(\mathbf{s}, a) = \mathbb{E}_\pi \left[R_t | \mathbf{s}_t = \mathbf{s}, a_t = a\right]$; the optimal Q-function is $Q^*(\mathbf{s}, a) = \max_\pi Q^\pi(\mathbf{s}, a)$.

The Bellman optimality equation writes Q^* recursively as

$$Q^*(\mathbf{s}, a) = \mathcal{T} Q^*(\mathbf{s}, a) \equiv r(\mathbf{s}, a) + \gamma \sum_{\mathbf{s}'} P(\mathbf{s}'|\mathbf{s}, a) \max_{a'} Q^*(\mathbf{s}', a'),$$

where P is the MDP state transition function and r is a reward function, which for simplicity we assume to be deterministic. Q-learning (Watkins & Dayan, 1992) uses a single-sample approximation of the contraction operator $\mathcal{T}$ to iteratively improve an estimate of Q^*.

In deep Q-learning (Mnih et al., 2015), Q is represented by a deep neural network with parameters θ, and is improved by regressing $Q(\mathbf{s}, a)$ to a target $r + \gamma \max_{a'} Q(\mathbf{s}', a'; \theta^-)$, where θ^- are the parameters of a target network periodically copied from θ.

We use a version of n-step Q-learning (Mnih et al., 2016) with synchronous environment threads. In particular, starting at a timestep t, we roll forward $n_{\text{env}} = 16$ threads for $n = 5$ timesteps each. We then bootstrap off the final states only and gather all $n_{\text{env}} \times n = 80$ transitions in a single batch for the backward pass, minimising the loss:

$$\mathcal{L}_{\text{nstep-Q}} = \sum_{\text{envs}} \sum_{j=1}^{n} \left(\sum_{k=1}^{j} \left[\gamma^{j-k} r_{t+n-k}\right] + \gamma^j \max_{a'} Q\left(\mathbf{s}_{t+n}, a', \theta^-\right) - Q\left(\mathbf{s}_{t+n-j}, a_{t+n-j}, \theta\right) \right)^2. \tag{1}$$

If the episode terminates, we use the remaining episode return as the target, without bootstrapping.

This algorithm's actor-critic counterpart is A2C, a synchronous variant of A3C (Mnih et al., 2016) in which a policy π and state-value function $V(s)$ are trained using the gradient:

$$\Delta\theta = \sum_{\text{envs}} \sum_{j=1}^{n} \left[\nabla_{\theta_\pi} \left(\log \pi(a_{t+j}|s_{t+j}) A_j(s_{t+j}, a_{t+j}) + \beta H(\pi(s_{t+j}))\right) + \alpha \nabla_{\theta_V} A_j(s_{t+j}, a_{t+j})^2 \right], \tag{2}$$

where A_j is an advantage estimate given by $\sum_{k=1}^{j} \gamma^{j-k} r_{t+n-k} + \gamma^j V(\mathbf{s}_{t+n}) - V(\mathbf{s}_{t+n-j})$, H is the policy entropy, β is a hyperparameter tuning the degree of entropy regularisation, and α is a hyperparameter controlling the relative learning rates of actor and critic.

These algorithms were chosen for their simplicity and reasonable wallclock speeds, but TreeQN can also be used in other algorithms, as described in Section 3.2. Our implementations are based on OpenAI Baselines (Hesse et al., 2017).

The canonical neural network architecture in deep RL with visual observations has a series of convolutional layers followed by two fully connected layers, where the final layer produces one output for each action-value. We can think of this network as first calculating an encoding $\mathbf{z}_t$ of the state $\mathbf{s}_t$ which is then evaluated by the final layer to estimate $Q^*(\mathbf{s}_t, a)$ (see Fig. 1).

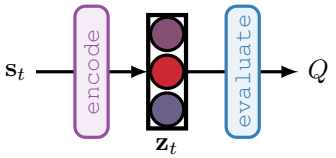


Figure 1: High-level structure of DQN.

In tree-search on-line planning, a look-ahead tree of possible future states is constructed by recursively applying an environment model. These states are typically evaluated by a heuristic, a learned value function, or Monte-Carlo rollouts. Backups through the tree aggregate these values along with the immediate rewards accumulated along each path to estimate the value of taking an action in the current state. This paper focuses on a simple tree-search with a deterministic transition function and no value uncertainty estimates, but our approach can be extended to tree-search variants like UCT (Kocsis & Szepesvári, 2006; Silver et al., 2016) if the components remain differentiable.

3 TREEQN

In this section, we propose TreeQN, a novel end-to-end differentiable architecture for tree planning. We first give an overview of the architecture, followed by details of each model component and the training procedure.

TreeQN uses a recursive tree-structured model between the encoded state $\mathbf{z}_t$ and the predicted state-action values $Q(\mathbf{s}_t, a)$. Instead of directly estimating the state-action value from the current encoded state $\mathbf{z}_t$ using fully connected layers as in DQN (Mnih et al., 2015) (Fig. 1), TreeQN uses a recursive model to refine its estimate of $Q(\mathbf{s}_t, a)$ via learned transition, reward, and value functions (see Fig. 2). Because these learned components are shared throughout the tree, TreeQN implements an inductive bias, missing from DQN, that reflects the prior knowledge that the Q-values are properties of a stationary Markov process.

Specifically, TreeQN learns an action-dependent transition function that, given a state representation $\mathbf{z}_t$, predicts the next state representation $\mathbf{z}_{t+1}^{a_i}$ for action $a_i \in \mathcal{A}$, and the corresponding reward $\hat{r}_t^{a_i}$. TreeQN applies this transition function recursively to construct a tree containing the state representations and rewards received for all possible sequences of actions up to some predefined depth d ("Tree Transitioning" in Fig. 2).

The value of each predicted state $V(\mathbf{z})$ is estimated with a value function module. Using these values and the predicted rewards, TreeQN then performs a tree backup, mixing the k-step returns along each path in the tree using TD(λ) (Sutton, 1988; Sutton & Barto, 1998). This corresponds to "Value Prediction" in Fig. 2.

$$Q^l(\mathbf{z}_{t+l}, a_i) = r(\mathbf{z}_{t+l}, a_i) + \gamma V^{(\lambda)}(\mathbf{z}_{t+l+1}) \tag{3}$$

$$V^{(\lambda)}(\mathbf{z}_{t+l}) = \begin{cases} V(\mathbf{z}_{t+l}^{a_i}) & l = d \\ (1-\lambda)V(\mathbf{z}_{t+l}^{a_i}) + \lambda \max_{a_j} Q^{l+1}(\mathbf{z}_{t+l+1}^{a_i}, a_j) & l < d \end{cases} \tag{4}$$

For $0 < \lambda < 1$, value estimates of the intermediate states are mixed into the final Q-estimate, which encourages the intermediate nodes of the tree to correspond to meaningful states, and reduces the impact of outlier values.

When $\lambda = 1$, Eq. 3 simplifies to a backup through the tree using the Bellman equation:

$$Q(\mathbf{z}_{t+l}, a_i) = r(\mathbf{z}_{t+l}, a_i) + \begin{cases} \gamma V(\mathbf{z}_{t+d}^{a_i}) & l = d \\ \gamma \max_{a_j} Q(\mathbf{z}_{t+l+1}^{a_i}, a_j) & l < d \end{cases} \tag{5}$$

We note that even for a tree depth of only one, TreeQN imposes a significant structure on the value function by decomposing it as a sum of action-conditional reward and next-state value, and using a shared value function to evaluate each next-state representation.

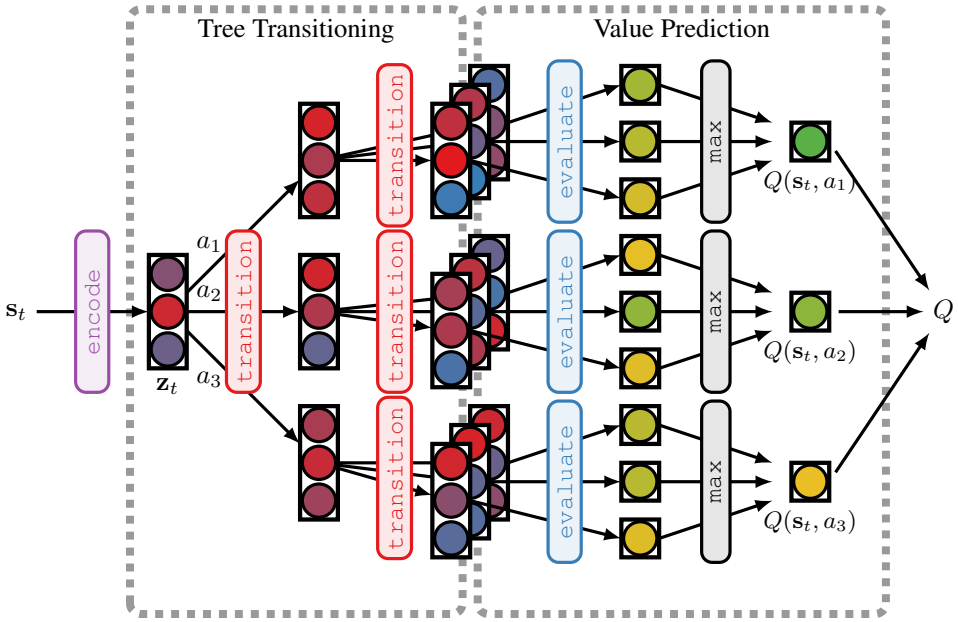


Figure 2: High-level structure of TreeQN with a tree depth of two and shared transition and evaluation functions (reward prediction and value-mixing omitted for simplicity).

Crucially, during training we backpropagate all the way from the final Q-estimate, through the value prediction, tree transitioning, and encoding layers of the tree, *i.e.*, the entire network shown in Fig. 2. Learning these components jointly ensures that they are useful for planning on-line.

3.1 Model Components

In this section, we describe each of TreeQN's components in more detail.

Encoder function. As in DQN, a series of convolutional layers produces an embedding of the observed state, $\mathbf{z}_t = \texttt{encode}(\mathbf{s}_t)$.

Transition function. Every action is parameterised by a fully connected layer. We use residual connections (He et al., 2016) to calculate a next-state representation that carries information about the effect of taking action $a_i \in \mathcal{A}$:

$$\mathbf{z}_{t+1}^{a_i} = \mathbf{z}_t + \tanh(\boldsymbol{W}^{a_i}\mathbf{z}_t), \tag{6}$$

where $\boldsymbol{W}^{a_i} \in \mathbb{R}^{k\times k}$ is a learnable action-dependent transition matrix. The next-state representation is calculated for every action independently, but the transition function is shared for the same action throughout the tree.

A caveat is that the model can still learn to use different parts of the latent state space in different parts of the tree, which could undermine the intended parameter sharing in the model structure. To help TreeQN learn useful transition functions that maintain quality and diversity in their plans, we introduce a unit-length projection of the state representations by simply dividing a state's vector representation by its L2 norm before each application of the transition function, $\mathbf{z}_t := \mathbf{z}_t / \|\mathbf{z}_t\|$. This prevents the magnitude of the representation from growing or shrinking, which encourages the behaviour of the transition function to be more consistent through the tree.

Reward function. In addition to predicting the next state, we also predict the immediate reward for every action $a_i \in \mathcal{A}$ in state $\mathbf{z}_t$ using

$$\hat{\mathbf{r}}(\mathbf{z}_t) = \boldsymbol{W}_2^r \texttt{ReLU}(\boldsymbol{W}_1^r \mathbf{z}_t + \mathbf{b}_1^r) + \mathbf{b}_2^r, \tag{7}$$

where $\boldsymbol{W}_1^r \in \mathbb{R}^{k\times k}$, $\boldsymbol{W}_2^r \in \mathbb{R}^{|\mathcal{A}|\times k}$ and $\texttt{ReLU}$ is the rectified linear unit (Nair & Hinton, 2010), and the predicted reward for a particular action $\hat{r}_t^{a_i}$ is the i-th element of the vector $\hat{\mathbf{r}}(\mathbf{z}_t)$.

Value function. The value of a state representation $\mathbf{z}$ is estimated with

$$V(\mathbf{z}) = \mathbf{w}^\top \mathbf{z} + b, \tag{8}$$

where $\mathbf{w} \in \mathbb{R}^k$.

3.2 Training

The TreeQN architecture is fully differentiable, so we can directly use it in the place of a Q-function in any deep RL algorithm with discrete actions. Differentiating through the entire tree ensures that the learned components are useful for planning on-line, as long as that planning is performed in the same way as during training.

While training, we also use an auxiliary reward-prediction loss, in the form of an L2 loss regressing $\hat{r}_1^a(\mathbf{z}_{t+j})$, the predicted reward at the first level of the tree corresponding to the selected action a, to the true observed reward, for each of the n timesteps of n-step Q-learning:

$$\mathcal{L} = \mathcal{L}_{\text{nstep-Q}} + \eta \sum_{\text{envs}} \sum_{j=1}^{n} (\hat{r}_1^a(\mathbf{z}_{t+j}) - r_{t+j})^2, \tag{9}$$

where η is a hyperparameter weighting the loss. In our experiments we simply use $\eta = 1$, and we generally found the performance to be robust with respect to η, though this may be a consequence of the well behaved reward scaling in our environments. The predicted rewards that this objective encourages the model to learn appear both in its own Q-value prediction and in the target for n-step Q-learning. Consequently, we expect this auxiliary loss to be well aligned with the true objective. By contrast, other potential auxiliary losses, *e.g.*, an observation-prediction loss, might help representation learning but would not explicitly learn any part of the desired target.

4 ATreeC

The intuitions guiding the design of TreeQN are as applicable to policy search as to value-based RL, in that a policy can use a tree planner to improve its estimates of the optimal action probabilities (Gelly & Silver, 2007; Silver et al., 2017a). As our proposed architecture is trained end-to-end, it can be easily adapted for use as a policy network.

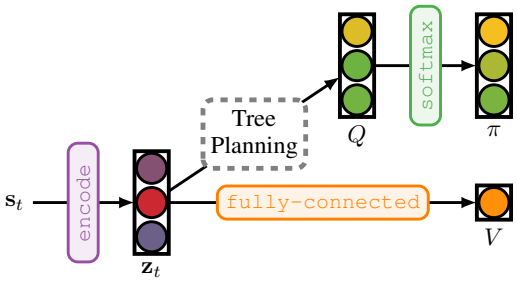


Figure 3: High-level structure of ATreeC.

In particular, we propose ATreeC, an actor-critic extension of TreeQN. In this architecture, the policy network is identical to TreeQN, with an additional softmax layer that converts the Q estimates into the probabilities of a stochastic policy. The critic shares the encoder parameters, and predicts a scalar state value with a single fully connected layer: $V_{\text{cr}}(\mathbf{s}) = \mathbf{w}_{\text{cr}}^\top \mathbf{z} + b_{\text{cr}}$. The entire setup, shown in Fig. 3, is trained with A2C as described in Section 2, with the addition of the same reward-prediction loss used for TreeQN. TreeQN could also be used in the critic, but we leave this possibility to future work.

5 Related Work

There is a long history of work combining model-based and model-free RL. An early example is Dyna-Q (Sutton, 1990) which trains a model-free algorithm with samples drawn from a learned model. Similarly, van Seijen et al. (2011) train a sparse model with some environment samples that can be used to refine a model-free Q-function. Gu et al. (2016) use local linear models to generate additional samples for their model-free algorithm. However, these approaches do not attempt to use the model on-line to improve value estimates.

In deep RL, *value iteration networks* (Tamar et al., 2016) use a learned differentiable model to plan on the fly, but require planning over the full state space, which must also possess a spatial structure with local dynamics such that convolution operations can execute the planning algorithm.

The *predictron* (Silver et al., 2017b) instead learns abstract-state transition functions in order to predict values. However, it is restricted to policy evaluation without control. *Value prediction networks* (VPNs) (Oh et al., 2017) take a similar approach but are more closely related to our work because the learned model components are used in a tree for planning. However, this tree is only used to construct targets and choose actions, and not to compute the value estimates during training. Such estimates are instead produced from non-branching trajectories following on-policy action sequences. By contrast, TreeQN is a unified architecture that constructs the tree dynamically at every timestep and differentiates through it, eliminating any mismatch between the model at training and test time. Furthermore, we do not use convolutional transition functions, and hence do not impose spatial structure on the latent state representations. These differences simplify training, allow our model to be used more flexibly in other training regimes, and explain in part our substantially improved performance on the Atari benchmark.

Donti et al. (2017) propose differentiating through a stochastic programming optimisation using a probabilistic model, to learn model parameters with respect to their true objective rather than a maximum-likelihood surrogate. However, they do not tackle the full RL setting, and do not use the model to repeatedly or recursively refine predictions.

Imagination-augmented agents (Weber et al., 2017) learn to improve policies by aggregating rollouts predicted by a model. However, they rely on pretraining an observation-space model, which we argue will scale poorly to more complex environments. Further, their aggregation of rollout trajectories takes the form of a generic RNN rather than a value function and tree backup, so the inductive bias based on the structure of the MDP is not explicitly present.

A class of *value gradient* methods (Deisenroth & Rasmussen, 2011; Fairbank & Alonso, 2012; Heess et al., 2015) also differentiates through models to train a policy. However, this approach does not use the model during execution to refine the policy, and requires continuous action spaces.

Oh et al. (2015) and Chiappa et al. (2017) propose methods for learning observation-prediction models in the Atari domain, but apply these models only to improve exploration. Variants of scheduled sampling (Bengio et al., 2015) may be used to improve robustness of these models, but scaling to complex domains has proven challenging (Talvitie, 2014).

6 Experiments

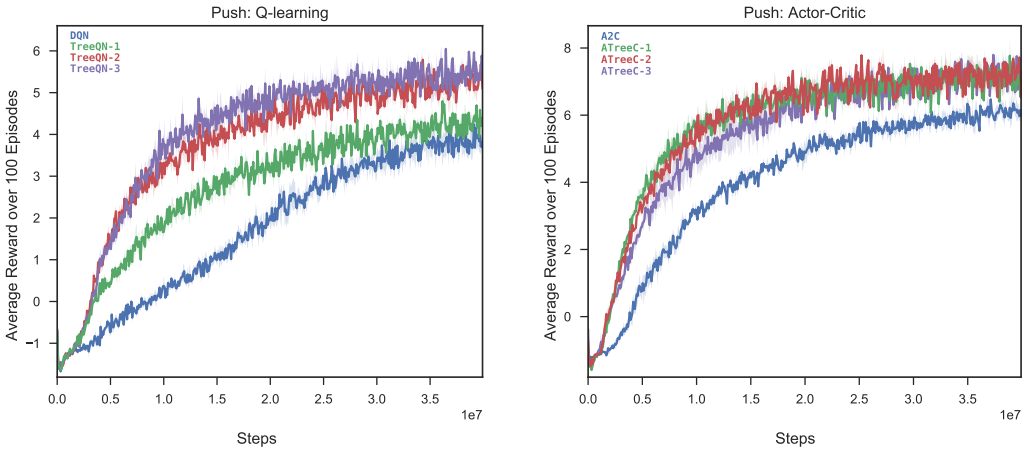
We evaluate TreeQN and ATreeC in a simple box-pushing environment, as well as on the subset of nine Atari environments that Oh et al. (2017) use to evaluate VPN. The experiments are designed to determine whether or not TreeQN and ATreeC outperform DQN, A2C, and VPN, and whether they can scale to complex domains. We also investigate the impact of tree depth, and the extent to which the trees represent meaningful plans. Full details of the experimental setup, as well as architecture and training hyperparameters, are given in the appendix.

Box Pushing. First, we experiment in a simple box-pushing domain. We randomly place an agent, 12 boxes, 5 goals and 6 obstacles on the center 6×6 tiles of an 8×8 grid. Figure 5a shows a typical state. The agent's goal is to push boxes into goals in as few steps as possible while avoiding obstacles. Boxes may not be pushed into each other. The obstacles, however, are 'soft' in that they are do not block movement, but generate a negative reward if the agent or a box moves onto an obstacle. This rewards better planning without causing excessive gridlock. This environment is inspired by Sokoban, as used by Weber et al. (2017), in that poor actions can generate irreversibly bad configurations. However, the level generation process for Sokoban is challenging to reproduce exactly and has not been open-sourced. More details of the environment and rewards are given in Appendix A.1.

Atari. To demonstrate the general applicability of TreeQN and ATreeC to complex environments, we evaluate them on the Atari 2600 suite (Bellemare et al., 2013). Following Oh et al. (2017), we use their set of nine environments and a frameskip of 10 to facilitate planning over reasonable timescales.

7 Results & Discussion

In this section, we present our experimental results for TreeQN and ATreeC, as well as a qualitative analysis of learned trees for the box-pushing environment.



(a) DQN and TreeQN. **(b)** A2C and ATreeC.

Figure 4: Results for the box-pushing domain. The x-axis shows the number of transitions observed across all of the synchronous environment threads.

7.1 Box Pushing

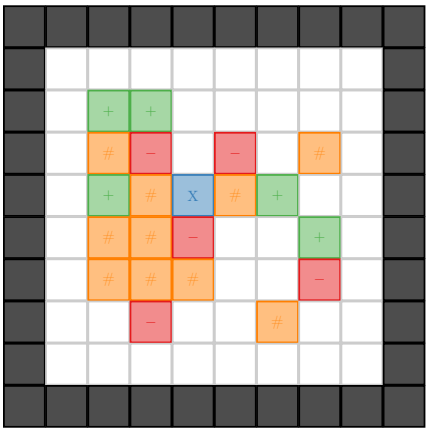
Fig. 4a shows the results of TreeQN with tree depths 1, 2, and 3, compared to a DQN baseline. In this domain, there is a clear advantage for the TreeQN architecture over DQN. TreeQN learns policies that are substantially better at avoiding obstacles and lining boxes up with goals so they can be easily pushed in later. TreeQN also substantially speeds up learning. We hypothesise that the greater structure brought by our architecture regularises the model, encouraging appropriate state representations to be learned quickly. Even a depth-1 tree improves performance significantly, as disentangling the esimation of rewards and next-state values makes them easier to learn. This is further facilitated by the sharing of value-function parameters across branches.

When trained with n-step Q-learning, there is a clear difference in the quality of TreeQN policies of different depths: deeper trees learn faster and plateau higher. In the box-pushing domain, useful transition functions are relatively easy to learn, and the extra computation time with those transition modules can help refine value estimates, yielding advantages for additional depth.

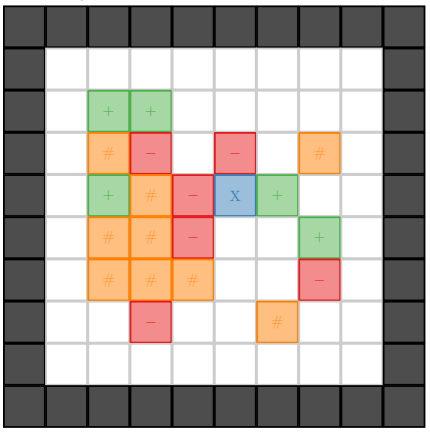
Fig. 4b shows the results of ATreeC with tree depths 1, 2, and 3, compared to an A2C baseline. As with TreeQN, ATreeC substantially outperforms the baseline. Furthermore, thanks to its stochastic policy, it substantially outperforms TreeQN. Whereas TreeQN and DQN sometimes indecisively bounce back and forth between adjacent states, ATreeC captures this uncertainty in its policy probabilities and thus acts more decisively. However, unlike TreeQN, ATreeC shows no pronounced differences for different tree depths. We hypothesise that this is due to a ceiling effect in this domain. Nonetheless, this result demonstrates the ease with which TreeQN can be used as a drop-in replacement for any deep RL algorithm that learns policies or value functions for discrete actions.

To shed further light on how TreeQN works, we can also inspect the learned policies and trees, whose values often correspond to intuitive reasoning about sensible policies. Fig. 5 shows one example. In Fig. 5a, the agent can push a box onto a goal by moving either left or right. Fig. 5c shows the corresponding tree values: each node contains the λ-mixed value of the path ending at that node, so the action chosen at each timestep is the first action in the highest value path. The two highest value paths are right-left, and left-right. Thus, TreeQN has learned that both outcomes are positive and that returning for the other box is the correct next move. Interestingly, the evaluation after the first level prefers to move left first, while the evaluation after the second level, after refining its values, prefers to move right first. This superior path allows the agent to quickly reach another box adjacent to a goal in the top left.

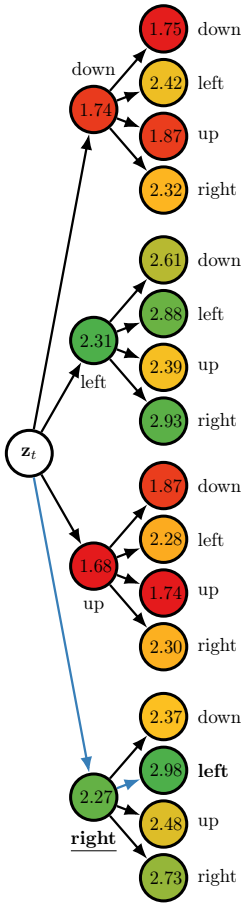
Fig. 5b shows the environment after the agent takes its action (move right) and Fig. 5d shows the corresponding tree. The agent now prefers to move left twice, scoring for pushing the second box.



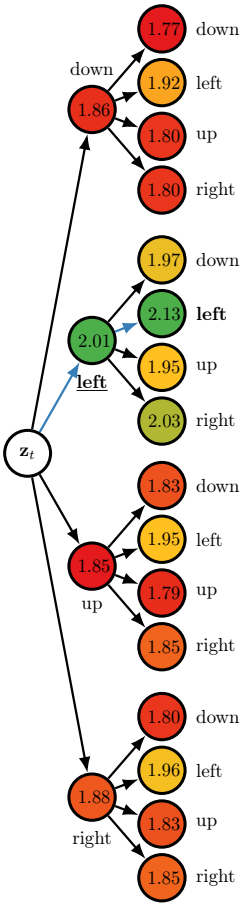
(a) The agent can score by pushing a box left or right.



(b) The agent must now traverse an obstacle to the left in order to push in another box.



(c) Tree for (a).



(d) Tree for (b).

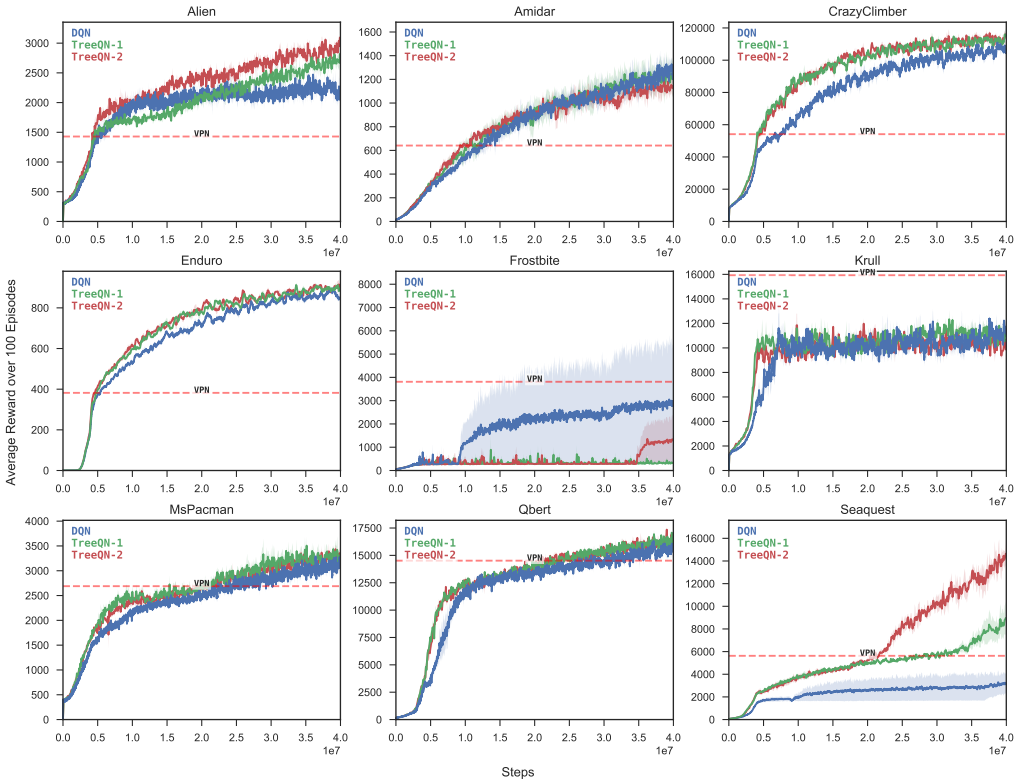
Figure 5: An example sequence of two states and corresonding value-trees as predicted by TreeQN, in which the values were easily interpretable (highest value path shown in blue). In the state diagrams, boxes are orange, obstacles are red, goals are green, and the agent is blue.

Although walking over the red obstacle incurs a small penalty (accounted for by the lower absolute values in the tree), the correct action sequence is planned and executed.

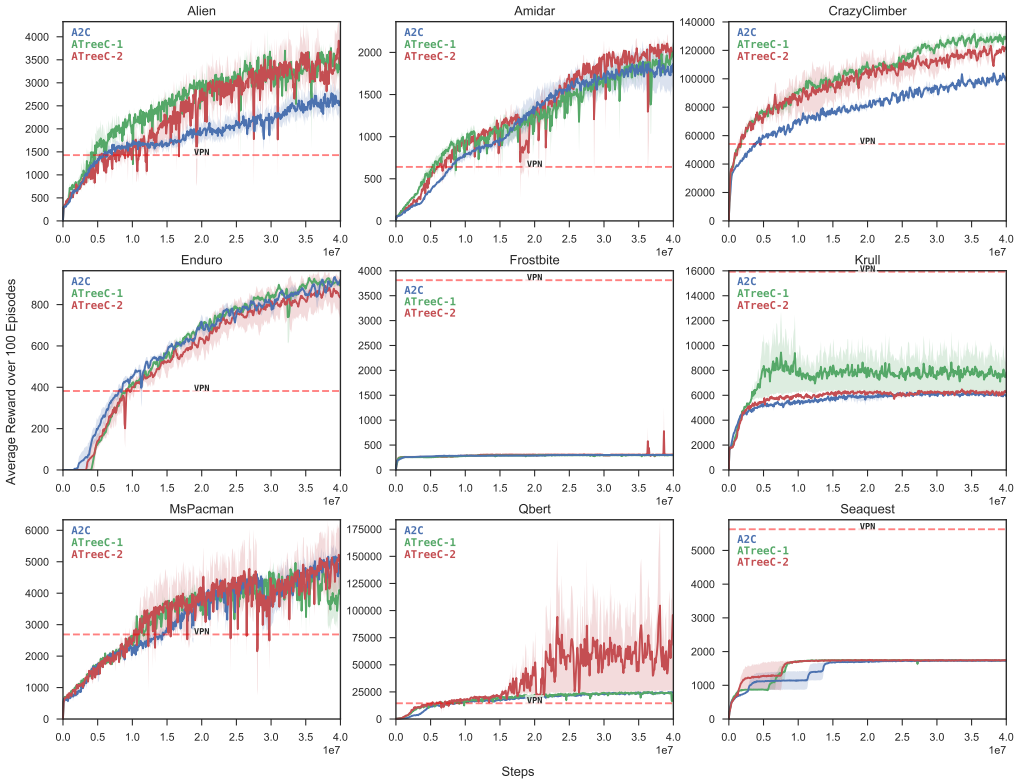
However, many of the planned action sequences do not correspond to the selected future actions. For example, the highest scoring path often consists of a repeated action, as in Fig. 5d, even if that is not the optimal sequence. We hypothesise that the agents use these paths to conduct extra action-conditional computations, which makes sense given that the levels in the tree are only weakly tied to the timesteps of the underlying MDP. Because TreeQN is flexible and trained end-to-end, it can learn to use its transition function in different ways depending on the circumstance. However, this flexibility can make the resulting plans less interpretable.

7.2 ATARI

Table 1 summarises all our Atari results, while Fig. 6 shows learning curves in depth. TreeQN shows substantial benefits in many environments compared to our DQN baseline, which itself often outperforms VPN (Oh et al., 2017). ATreeC also often outperforms A2C. We present the mean performance of three random seeds, while the VPN results reported by Oh et al. (2017), shown as dashed lines in Fig. 6, are the mean of the *best* five seeds of an unspecified number of trials.



(a) DQN and TreeQN trained with n-step Q-learning.



(b) A2C and ATreeC.

Figure 6: Results for the Atari domain.

	Alien	Amidar	Crazy Climber	Enduro	Frostbite	Krull	Ms. Pacman	Q*Bert	Seaquest
DQN (Oh et al., 2017)	1804	535	41658	326	3058	12438	2804	12592	2951
VPN (Oh et al., 2017)	1429	641	54119	382	**3811**	**15930**	2689	14517	5628
n-step DQN	2472	1329	110947	897	3054	12405	3290	16323	3271
TreeQN-1	2819	1271	116034	913	893	12344	3507	17082	8974
TreeQN-2	**3096**	1186	116779	915	1360	11980	3426	17355	**14654**
A2C	2776	1910	103866	**934**	305	6282	5172	24872	1739
ATreeC-1	3758	2065	**131655**	931	302	11067	4964	25208	1753
ATreeC-2	2917	**2106**	123257	891	781	6485	**5225**	**104657**	1756

Table 1: Summary of Atari results. Each number is the best score throughout training, calculated as the mean of the last 100 episode rewards of three agents trained with different random seeds. Note that Oh et al. (2017) report the same statistic, but average instead over the *best* 5 of an unspecified number of agents.

TreeQN. In Alien, CrazyClimber, and Seaquest, TreeQN significantly outperforms the baselines, while we see smaller advantages for Enduro, MsPacman and Qbert. Amidar and Krull show no significant differences between methods, and Frostbite has degenerately poor performance for all methods. Unsurprisingly, TreeQN performs well in domains that are suited to look-ahead planning, with simple dynamics that generalise well and tradeoffs between actions that become apparent only after several timesteps. For example, an incorrect action in Alien can trap the agent down a corridor with an alien. In Seaquest, planning ahead can help determine whether it is better to go deeper to collect more points or to surface for oxygen. By contrast, Enduro is a racing game with mostly reactive decisions that can be easily adjusted in subsequent timesteps.

ATreeC. ATreeC, like TreeQN, outperforms its baseline strongly in Alien and Crazy Climber. Compared to TreeQN, ATreeC's performance is better across most environments, particularly on Qbert, reflecting an overall advantage for actor-critic also found by Mnih et al. (2016). Performance is worse on Seaquest, revealing a deficiency in exploration as policy entropy collapses too rapidly.

In environments like Krull and Frostbite, all agents are clearly gated by their ability to explore. Both of these games require the completion of sub-levels in order to accumulate large scores, and none of our agents reliably explore beyond the initial stage of the game. Combining TreeQN and ATreeC with smart exploration mechanisms is an interesting direction for future work.

Compared to the box-pushing domain, there is less of a clear performance difference between trees of different depths. In some environments (Alien, Seaquest), greater depth does appear to be employed usefully by TreeQN. However, for the Atari domain the embedding size for the transition function we use is much larger (512 compared to 128), and the dynamics are much more complex. Consequently, we expect that optimisation difficulties, and the challenge of learning abstract-state transition functions, impede the utility of deeper trees in some cases. We look to future work to further refine methods for learning to plan abstractly in complex domains.

8 Conclusions & Future Work

We presented TreeQN and ATreeC, new architectures for deep reinforcement learning in discrete-action domains that integrate differentiable on-line tree planning into the action-value function or policy. Experiments on a box-pushing domain and a set of Atari games show the benefit of these architectures over their model-free counterparts, as well as over VPN. In future work we intend to investigate enabling more efficient optimisation of deeper trees, encouraging the transition functions to produce interpretable plans, and integrating smart exploration. Stochastic transition functions for non-deterministic environments are also an interesting avenue for research.

References

Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *J. Artif. Intell. Res.(JAIR)*, 47:253–279, 2013.

Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. Scheduled sampling for sequence prediction with recurrent neural networks. In *Advances in Neural Information Processing Systems*,

pp. 1171–1179, 2015.

Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

Silvia Chiappa, Sébastien Racaniere, Daan Wierstra, and Shakir Mohamed. Recurrent environment simulators. *arXiv preprint arXiv:1704.02254*, 2017.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, pp. 72–83, 2006. doi: 10.1007/978-3-540-75538-8_7.

Marc Deisenroth and Carl E Rasmussen. Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pp. 465–472, 2011.

Priya L Donti, Brandon Amos, and J Zico Kolter. Task-based end-to-end model learning. *arXiv preprint arXiv:1703.04529*, 2017.

Michael Fairbank and Eduardo Alonso. Value-gradient learning. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pp. 1–8. IEEE, 2012.

Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pp. 273–280. ACM, 2007.

Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pp. 2829–2838, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pp. 770–778, 2016. doi: 10.1109/CVPR.2016.90.

Nicolas Heess, Gregory Wayne, David Silver, Tim Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pp. 2944–2952, 2015.

Christopher Hesse, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. `https://github.com/openai/baselines`, 2017.

Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6 (4):293–326, 1975.

Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, pp. 282–293, 2006. doi: 10.1007/11871842_29.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. doi: 10.1038/nature14236.

Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 1928–1937, 2016.

Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pp. 807–814, 2010.

Junhyuk Oh, Xiaoxiao Guo, Honglak Lee, Richard L Lewis, and Satinder Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in Neural Information Processing Systems*, pp. 2863–2871, 2015.

Junhyuk Oh, Satinder Singh, and Honglak Lee. Value prediction network. *arXiv preprint arXiv:1707.03497*, 2017.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *arXiv preprint arXiv:1705.05363*, 2017.

David Silver, Richard S. Sutton, and Martin Müller. Temporal-difference search in computer go. *Machine Learning*, 87(2):183–219, 2012. doi: 10.1007/s10994-012-5280-0.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017a.

David Silver, Hado van Hasselt, Matteo Hessel, Tom Schaul, Arthur Guez, Tim Harley, Gabriel Dulac-Arnold, David P. Reichert, Neil Rabinowitz, André Barreto, and Thomas Degris. The predictron: End-to-end learning and planning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pp. 3191–3199, 2017b.

Nathan R Sturtevant. An analysis of uct in multi-player games. In *International Conference on Computers and Games*, pp. 37–49. Springer, 2008.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3 (1):9–44, 1988.

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning, Proceedings of the Seventh International Conference on Machine Learning, Austin, Texas, USA, June 21-23, 1990*, pp. 216–224, 1990.

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

Erik Talvitie. Model regularization for stable sample rollouts. In *UAI*, pp. 780–789, 2014.

Erik Talvitie. Self-correcting models for model-based reinforcement learning. In *AAAI*, pp. 2597–2603, 2017.

Aviv Tamar, Sergey Levine, Pieter Abbeel, Yi Wu, and Garrett Thomas. Value iteration networks. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pp. 2146–2154, 2016.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Harm van Seijen, Shimon Whiteson, Hado van Hasselt, and Marco Wiering. Exploiting best-match equations for efficient reinforcement learning. *Journal of Machine Learning Research*, 12(Jun): 2045–2094, 2011.

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992. doi: 10.1007/BF00992698.

Theophane Weber, Sébastien Racanière, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adrià Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning. *CoRR*, abs/1707.06203, 2017.

A APPENDIX

A.1 BOX PUSHING

Environment. For each episode a new level is generated by placing an agent, 12 boxes, 5 goals and 6 obstacles in the center 6×6 tiles of an 8×8 grid, sampling locations uniformly. The outer tiles are left empty to prevent situations where boxes cannot be recovered.

The agent may move in the four cardinal directions. If the agent steps off the grid the episode ends and it receives a penalty of -1. If the agent moves into a box, it is pushed in the direction of movement. Moving a box out of the grid generates a penalty of -0.1. Moving a box into another box is not allowed and trying to do so generates a penalty of -0.1 while leaving all positions unchanged. When a box is pushed into a goal, it is removed and the agent receives a reward of +1.

Obstacles generate a penalty of -0.2 when the agent or a box is moved onto them. Moving the agent over goals incurs no penalty. Lastly, at each timestep the agent receives a penalty of -0.01. Episodes terminate when 75 timesteps have elapsed, the agent has left the grid, or no boxes remain.

The observation is given to the model as a tensor of size $5 \times 8 \times 8$. The first four channels are binary encodings of the position of the agent, goals, boxes, and obstacles respectively. The final channel is filled with the number of timesteps remaining (normalised by the total number of timesteps allowed).

Architecture. The encoder consists of (conv-3x3-1-32, conv-3x3-1-32, conv-4x4-1-64, fc-128), where conv-wxh-s-n denotes a convolution with n filters of size $w \times h$ and stride s and fc-h denotes a fully connected layer with h hidden units. All layers are separated with ReLU nonlinearities. The hidden layer of the reward function MLP has 64 hidden units.

A.2 ATARI

Preprocessing of inputs follows the procedure of Mnih et al. (2015), including concatenation of the last four frames as input, although we use a frameskip of 10.

Architecture. The Atari experiments have the same architecture as for box-pushing, except for the encoder architecture which is as follows: (conv-8x8-4-32, conv-4x4-2-64, conv-3x3-1-64, fc-512).

A.3 OTHER HYPERPARAMETERS

All experiments use RMSProp (Tieleman & Hinton, 2012) with a learning rate of 1e-4, a decay of α=0.99, and ϵ=1e-5.

The learning rate was tuned coarsely by running DQN on the Seaquest environment, and kept the same for all subsequent experiments (box-pushing and Atari).

For DQN and TreeQN, ϵ for ϵ-greedy exploration was decayed linearly from 1 to 0.05 over the first 4 million environment transitions observed.

For A2C and ATreeC, we use a value-function loss coefficient $\alpha = 0.5$ and an entropy regularisation $\beta = 0.01$.

The reward prediction loss was scaled by $\eta = 1$.

We use $n_{\text{steps}} = 5$ and $n_{\text{envs}} = 16$, for a total batch size of 80.