Policy Tree Network

Zac Wellmer
Kelly Kan Huang

Abstract

This paper proposes a novel deep reinforcement learning architecture which builds on previous tree structured architectures to allow for taking full advantage of tree structure in both training and testing while maintaining full alignment of the tree structure during training. Our approach integrates model-free and model-based reinforcement learning. Our experiments are focused on the Breakout Atari environment so that they can be compared with similar work done in this area.

1 Introduction

Forward planning is a critical part of Model-based reinforcement learning and requires some sort of transition model in order to plan several steps forward into the future. The transition model allows the agent to imagine what the environment will look like multiple steps into the future. In the case of our paper we will only be focused on agent's with implicit transition models[8][9] while there has been some interesting results which involve transition models predicting the observation [6]. This is a challenging task because we have no underlying knowledge of what the optimal hidden state should be.

In previous works, the architectures either did not support taking full advantage of tree planning in training and testing[8] or had a strong possibility of being misaligned during training[9]. Here we take on both of these issues and show the empirical capabilities on the discrete control tasks found in atari [2].

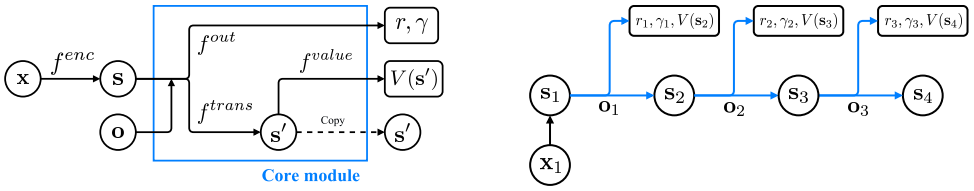
2 Related Work

Below we will give a brief review of work done in model based reinforcement learning, model free reinforcement learning, and traditional tree planning methods.

2.1 Model-based Reinforcement Learning

The essence of model based reinforcement learning revolves around using a model or learning a model of the environment and using this with a policy to plan optimal actions. A subclass of problems exist where the environment dynamics are fully known, for example the a deterministic instance of the classic gridworld problem. However, we will be focused on the class of problems where the environment dynamics are unknown and must be learned. In this case the problem can be broken down further into dynamics models that are learned implicitly and dynamics models that are learned explicitly. In the history of reinforcement learning explicitly learned dynamics models have been the focus of most attention. It was not until recent years that implicitly learned model based algorithms received attention [8] [9] [7].

Explicit Model-based Methods. Cases of explicit model-based methods involve directly predicting future observations and including this in the loss function as in [6]. This has seen some limited success and can sometimes provide a useful signal. This is particularly useful when observations contain entirely useful information. However, in the class of problems where the entire observation does not include useful information it is misleading. For example when generating a image frame



(a) One-step rollout (b) Multi-step rollout

Figure 1: [8]Both Value prediction network and Policy Tree Networks. (a) learns to predict immediate reward, and the value of the next abstract-state.(PTN does not learn discount) (b) unrolls the core module in the abstract-state space to compute multi-step rollouts.

a large part of the networks capacity will be dedicated to learning less useful information like the background or objects that are not of the agent's interest.

Implicit Model-based Methods. Implicit model-based methods are interesting because they are not explicitly referenced in the loss function. Rather they are learned by finding parameters that allow for an agent to perform optimally. This is the class of model based methods that our Policy Tree Networks fall under. Our work was inspired by the work of Value Prediction Networks[8] and ATreeC[9]. Value Prediction Networks involve expanding a Q-tree, performing a linear backup along the maximal path, and selecting the maximal backedup path. At training time the loss is computed along expanded tree path. ATreeC involves expanding a pseudo Q-tree. We call this a pseudo Q-tree because the nested value predictions are not actually constrained to represent the real value. The pseudo Q-values are treated as logits and are used to parameterize and sample from a Multinomial distribution. These samples are used as the actions. At training time the Q-tree is not constrained to the same path as the one constructed by the generated rollout which causes a distribution mismatch when minimizing estimates of subtree auxillary estimates(reward/value depending on configuration). Both of VPN and ATreeC are constrained to only operating in discrete action spaces.

2.2 Model-free Deep Reinforcement Learning

The Deep Q-Network [5] architecture blazed the trail for model free deep reinforcement learning. Marking the first agent able to excel at excel at a diverse array of challenging tasks. Proceeding the ground breaking DQN architecture there has been a plethora of variants introduced providing improvements [3]. These previous works require storing a large replay buffer and in some cases parameterizing a distribution over the replay buffer. When dealing with agents operating in a state space comprised of images, storing the past million observations, as is the case in DQN, can be expensive in terms of space. Our Policy Tree Networks do not require storing a large replay buffer.

3 Policy Tree Networks

Similar to the Value Prediction Network a Policy Tree Network decomposes a Q value prediction to predicting $r_t = (s_t, a_t)$, $s_{t+1} = f_{transition}(s_t, a_t)$, and $v_{t+1} = v(s_{t+1})$. This is then combined to become $Q_t = r_t + \gamma v_{t+1}$. A key feature here is that in the case of Policy Tree Networks is that v_t is not actually directly optimized to reflect that of the true value function. The reason for this is because the Q_t values are linearly backed up and are treated as logits in the actor parameterizes a multinomial to sample from for action selection. In Section 3.2, we describe how to perform planning using Policy Tree Networks. Section 3.3 describes how to train Policy Tree Networks in a Actor-Critic-like framework [4].

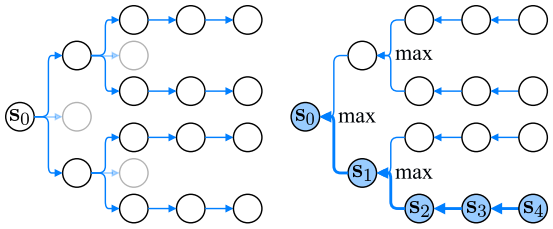
3.1 Architecture

The Policy Tree Network consists of the following modules parameterized by $\theta = \{\theta^{enc}, \theta^{value}, \theta^{reward}, \theta^{trans}, \theta^{critic}\}$:

Encoding $f_\theta^{enc} : \mathbf{x} \mapsto \mathbf{s}$ **Value** $f_\theta^{value} : \mathbf{s} \mapsto V_{\theta_a}(\mathbf{s})$

Reward $f_\theta^{reward} : \mathbf{s}, \mathbf{a} \mapsto r$ **Transition** $f_\theta^{transition} : \mathbf{s}, \mathbf{a} \mapsto \mathbf{s}'$ **Critic** $f_\theta^{critic} : \mathbf{s}, \mathbf{a} \mapsto V_{\theta_c}(\mathbf{s})$

- **Encoding** module maps the observation ($\mathbf{x}$) to the abstract state ($\mathbf{s} \in \mathbb{R}^m$) using neural networks (e.g., CNN for visual observations). Thus, $\mathbf{s}$ is an *abstract-state representation which will be learned by the network (and not an environment state or even an approximation to one).*



(a) Expansion (b) Backup

Figure 2: [8]Planning with Value Prediction Networks and Policy Tree Networks. (a) Simulate b-best options up to a certain depth ($b = 2$ in this example). (b) Aggregate all possible returns along the best sequence of future options.

Algorithm 1 Q-value from d-step planning

function Q-PLAN($\mathbf{s}, \mathbf{a}, d$)
  $r, V(\mathbf{s}'), \mathbf{s}' \leftarrow f_\theta^{core}(\mathbf{s}, \mathbf{a})$
  if $d = 1$ **then**
    return $r + \gamma V(\mathbf{s}')$
  end if
  $\mathcal{A} \leftarrow b$-best actions based on $Q^1(\mathbf{s}', \mathbf{a}')$
  for $\mathbf{a}' \in \mathcal{A}$ **do**
    $q_{\mathbf{a}'} \leftarrow$ Q-PLAN($\mathbf{s}', \mathbf{a}', d-1$)
  end for
  return $r + \gamma \left[\frac{1}{d} V(\mathbf{s}') + \frac{d-1}{d} \max_{\mathbf{a}' \in \mathcal{A}} q_{\mathbf{a}'}\right]$
end function

- **Value** module estimates the value of the abstract-state ($V_\theta(\mathbf{s})$). Note that the value module is not a function of the observation, but a function of the abstract-state.
- **Reward** module predicts the reward ($r \in \mathbb{R}$) for executing the action $\mathbf{a}$ at abstract-state $\mathbf{s}$
- **Transition** module transforms the abstract-state to the next abstract-state ($\mathbf{s}' \in \mathbb{R}^m$) by predicting $\Delta\mathbf{s} = \mathbf{s}' - \mathbf{s}$. Also note that the V_{θ_a} and V_{θ_c} do not share parameters.

Figure 1a illustrates the *core* module which performs 1-step rollout by composing the above modules: $f_\theta^{core} : \mathbf{s}, \mathbf{a} \mapsto r, V_\theta(\mathbf{s}'), \mathbf{s}'$. The core module takes an abstract-state and action as input and makes separate predictions of the abstract-state, reward, and the value of the abstract-state. By combining the predictions, we can estimate the pseudo Q-value as follows: $Q_\theta(\mathbf{s}, \mathbf{a}) = r + \gamma V_\theta(\mathbf{s}')$. In addition, the Policy Tree Network recursively applies the core module to predict the sequence of future abstract-states, rewards and values given and initial abstract-state as illustrated in Figure 1b.

3.2 Planning

Policy Tree Networks have the ability to predict the future abstract states and based on these predicted abstract states plan actions. Similar to VPN[8] we use a basic planning method which performs rollouts up to a certain depth d aggregating value estimates along the way as described in Algorithm 1 and Figure 2. Specifically, given an abstract state $\mathbf{s} = f_\theta^{enc}(\mathbf{x})$ and an action $\mathbf{a}$, the pseudo Q-value calculated from d-step planning is defined as:

$$Q_\theta^d(\mathbf{s}, \mathbf{a}) = r + \gamma V_{\theta_a}^d(\mathbf{s}') \qquad V_{\theta_a}^d(\mathbf{s}) = \begin{cases} V_{\theta_a}(\mathbf{s}) & \text{if } d = 1 \\ \frac{1}{d} V_{\theta_a}(\mathbf{s}) + \frac{d-1}{d} \max_{\mathbf{a}} Q_\theta^{d-1}(\mathbf{s}, \mathbf{a}) & \text{if } d > 1, \end{cases} \tag{1}$$

where $\mathbf{s}' = f_\theta^{trans}(\mathbf{s}, \mathbf{a}), V_\theta(\mathbf{s}) = f_\theta^{value}(\mathbf{s})$, and $r, = f_\theta^{reward}(\mathbf{s}, \mathbf{a})$. Our planning algorithm is divided into two steps: expansion and backup. At the expansion step (see Figure 2a), we recursively simulate options up to a depth of d by unrolling the core module. At the backup step, we compute the weighted average of the direct value estimate $V_\theta(\mathbf{s})$ and $\max_{\mathbf{a}} Q_\theta^{d-1}(\mathbf{s}, \mathbf{ac})$ to compute $V_\theta^d(\mathbf{s})$ (i.e., value from d-step planning) in Equation 1. Note that $\max_{\mathbf{action}} Q_\theta^{d-1}(\mathbf{s}, \mathbf{a})$ is the average over $d-1$ possible value estimates. We propose to compute the uniform average over all possible returns by using weights proportional to 1 and $d-1$ for $V_\theta(\mathbf{s})$ and $\max_{\mathbf{a}} Q_\theta^{d-1}(\mathbf{s}, \mathbf{a})$ respectively. Thus, $V_\theta^d(\mathbf{s})$ is the uniform average of d expected returns along the path of the best sequence of actions as illustrated in Figure 2b.

To reduce the computational cost, we simulate only b-best options at each expansion step based on $Q^d(\mathbf{s}, \mathbf{a})$ when $d > 1$. d must be > 1 because when we evaluate for $d = 0$ we need action probabilities to be nonzero for all possible actions.

3.3 Learning

Policy Tree Networks can be trained through any existing actor-critic based RL algorithm. In this paper, we use a modified version of n-step Q-learning [4]. The main idea is to generate trajectories by sampling actions from the backed up pseudo Q-values treated as logits. Given an n-step trajectory $\mathbf{x}_1, \mathbf{a}_1, r_1, \mathbf{x}_2, \mathbf{a}_2, r_2, ..., \mathbf{x}_{n+1}$ generated by the policy, k-step predictions are defined as follows:

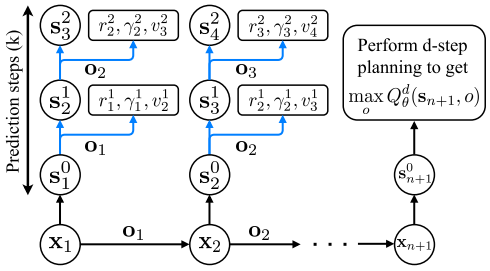


Figure 3: [8]Illustration of learning process.

$$\mathbf{s}_t^k = \begin{cases} f_\theta^{enc}(\mathbf{x}_t) & \text{if } k = 0 \\ f_\theta^{trans}(\mathbf{s}_{t-1}^{k-1}, \mathbf{a}_{t-1}) & \text{if } k > 0 \end{cases} \qquad v_t^k = f_\theta^{value_a}(\mathbf{s}_t^k) \qquad r_t^k = f_\theta^{reward}(\mathbf{s}_t^{k-1}, \mathbf{a}_t).$$

Intuitively, $\mathbf{s}_t^k$ is the Policy Tree Network's k-step prediction of the abstract-state at time t predicted from $\mathbf{x}_{t-k}$ following actions $\mathbf{a}_{t-k}, ..., \mathbf{a}_{t-1}$ in the trajectory as illustrated in Figure 3. By applying the value and the reward module, Policy Tree Network can compute the k-step prediction of the value and the reward. The k-step prediction loss at step t is defined as:

$$\mathcal{L}_t = -log\_prob(a_t) Advantage_t - \beta H + V_c(s_t) + \sum_{l=1}^{k} \left(r_t - r_t^l\right)^2$$

where $Advantage_t = -V_{c,\theta}(s_t) + \gamma^k V_{c,\theta^-}(s_{t+k}) + \sum_{i=0}^{k} \gamma^i r(s_{n+i}, a_{n+i})$ is the advantage estimate. Intuitively, $\mathcal{L}_t$ accumulates losses over 1-step to k-step predictions of rewards given fixed set of actions the agent actually followed to generate these target rewards in the rollout. Note that we do not enforce the trajectory on the subtree when generating logits to parameterize the distribution used for calculating $log_prob(a_t)$Then applies n-step learning for policy gradient estimates and critic value estimates. β is a scalar term applied to the entropy exploration bonus.

Our learning algorithm introduces two hyperparameters: the number of prediction steps (k) and planning depth (d_{train}) used for choosing options and computing bootstrapped targets. We also make use of a *target network* parameterized by θ^- which is synchronized with θ after a certain number of steps to stabilize training as suggested by [4]. The loss is accumulated over n-steps and the parameter is updated by computing its gradient as follows: $\nabla_\theta \mathcal{L} = \sum_{t=1}^{n} \nabla_\theta \mathcal{L}_t$.

3.4 Policy

Action selection in Policy Tree Networks is done similarly to that of ATreeC[9]. The backed up pseudo Q-tree estimates are treated as logits to parameterize a multinomial distribution. Specifically, the policy generates actions:

$$a_t \sim \pi(Q) \tag{2}$$

Where Q is the base of the backed up Q tree.

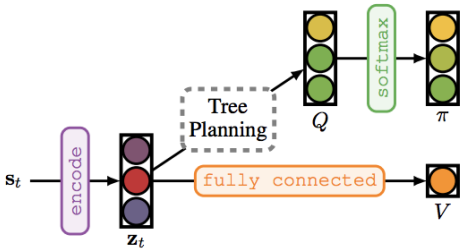


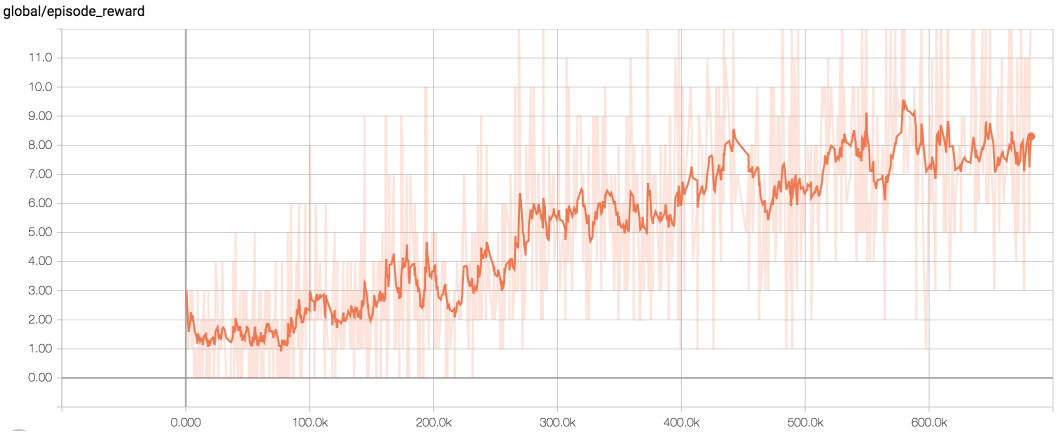
Figure 4: [9] Parameterizing the policy.

4 Experiments

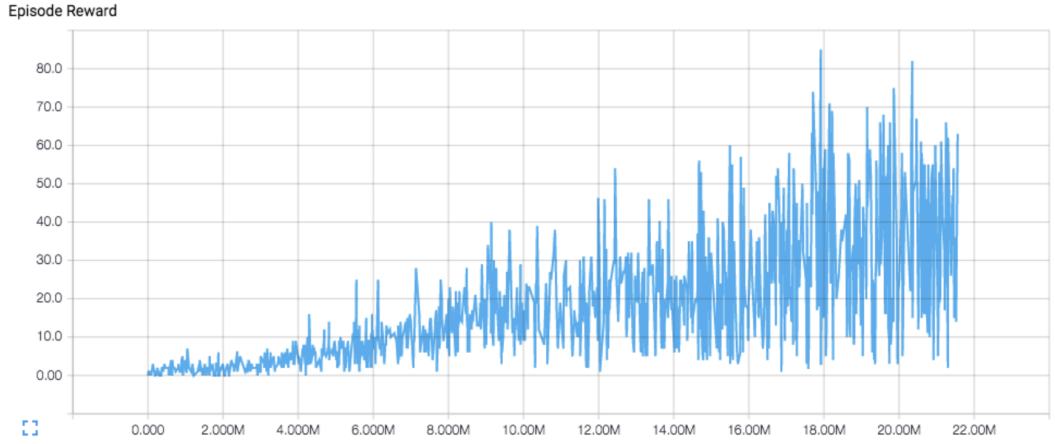
Our experiments investigated the comparison of Policy Tree Networks to A3C [4]. Due to computational restraints we were not able to run experiments as detailed as desired. Most experiments in the referenced papers had access to large GPU clusters with large number of CPU cores. We are working on running larger scale experiments which will compare results to VPN and ATreeC across a wider set of environments.

Table 1: Performance on Atari games. Each number represents average score over 5 top agents.

	Frostbite	Seaquest	Enduro	Alien	Q*Bert	Ms. Pacman	Amidar	Krull	Crazy Climber
DQN	3058	2951	326	1804	12592	2804	535	12438	41658
PTN	**?**	?	?	?	?	?	?	?	?



(a) Policy Tree Network Preliminary Breakout Results



(b) A3C Breakout Results

4.1 Atari Games

To test our model we chose to use the Breakout simulator from the atari environment [2]. Unfortunately our tests are extremely limited due to computation constraints. We preprocessed the frames to 84×84 gray-scale images. And we take the last 4 frames as input.

Due to the limited tests it is difficult to make any conclusive decisions about the architecture. However, it is easy to see that the Policy Tree Network is learning at a cheaper sample complexity than the A3C [4]. Notice the Policy Tree Network reaches an average episode reward of 8.0 at around 400k iterations. Where A3C does not appear to pass 8.0 until around 4 million iterations.

5 Conclusion

Introduced in this work is a new way of doing forward planning over abstract states without a distribution mismatch in auxiliary rewards found in[9] and also keeping the benefits of tree planning at both training and test time found in ATreeC but missing from VPN. The preliminary results seem interesting but further tests need to be run on larger compute instances.

Notes

This work was built on top of the universe starter agent and primarily uses tensorflow framework[1]. Some figures in this paper were taken directly from VPN and TreeQN.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *arXiv preprint arXiv:1207.4708*, 2012.

[3] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable MDPs. *arXiv preprint arXiv:1507.06527*, 2015.

[4] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *ICML*, 2016.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[6] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh. Action-conditional video prediction using deep networks in atari games. In *NIPS*, 2015.

[7] D. Silver, H. van Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto, and T. Degris. The predictron: End-to-end learning and planning. In *ICML*, 2017.

[8] J. Oh, S. Singh., and H. Lee. Value Prediction Networks. In *NIPS*, 2017.

[9] G. Farquhar, T. Rocktaschel, M. Igl, S. Whiteson TreeQN and ATreeC: Differentiable Tree-Structured Models for Deep Reinforcement Learning. In *ICLR*, 2018.