7-2021

Dynamic Planning Networks

Norman Tasfi
Western University, ntasfi@uwo.ca

Miriam A M Capretz
Western University, mcapretz@uwo.ca

Dynamic Planning Networks

Norman Tasfi & Miriam Capretz
Department of Electrical And Computer Engineering
Western University
London, Ontario, Canada
{ntasfi, mcapretz}@uwo.ca

***Abstract*—We introduce Dynamic Planning Networks (DPN), a novel architecture for deep reinforcement learning, that combines model-based and model-free aspects for online planning. Our architecture learns to dynamically construct plans using a learned state-transition model by selecting and traversing between simulated states and actions to maximize information before acting. DPN learns to efficiently form plans by expanding a single action-conditional state transition at a time instead of exhaustively evaluating each action, reducing the number of state-transitions used during planning. We observe emergent planning patterns in our agent, including classical search methods such as breadth-first and depth-first search. DPN shows improved performance over existing baselines across multiple axes.**

***Index Terms*—deep neural networks, reinforcement learning, planning**

I. Introduction

The central focus of reinforcement learning (RL) is the selection of optimal actions to maximize the expected reward in an environment where the agent must rapidly adapt to new and varied scenarios. Various avenues of research have spent considerable efforts improving core axes of RL algorithms such as performance, stability, and sample efficiency. Significant progress on all fronts has been achieved by developing agents using deep neural networks with model-free RL [1]–[4]; showing model-free methods efficiently scale to high-dimensional state space and complex domains with increased compute. Unfortunately, model-free policies are often unable to generalize to variances within an environment as the agent learns a policy which directly maps environment states to actions. A favorable approach to improving generalization is to combine an agent with a learned environment model, enabling it to reason about its environment. This approach, referred to as model-based RL learns a model from past experience, where the model usually captures state-transitions, $p(s_{t+1}|s_t, a_t)$, and might also learn reward predictions $p(r_{t+1}|s_t, a_t)$. Usage of learned state-transition models is especially valuable for planning, where the model predicts the outcome of proposed actions, avoiding expensive trial-and-error in the actual environment – improving performance and generalization. This contrasts with model-free methods which are explicitly trial-and-error learners [5]. Historically, applications have primarily focused on domains where a state-transition model can be easily learned, such as low dimensional observation spaces [6]–[8], or where a perfect model was provided [9], [10] – limiting usage.

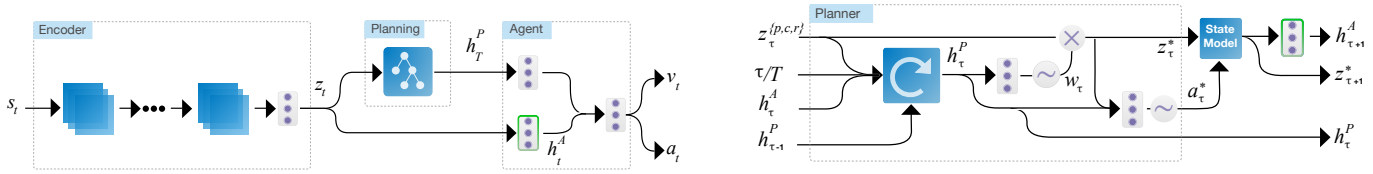
The planning style, how the state-transition model is applied, is an important factor to consider as it can affect the efficiency of the simulated trajectory. Typically, the planning style is explicitly set per architecture, with various styles used such as: recursively expanding all available actions per state for a fixed depth [4], [11], expanding all actions of the initial state and simulating forward for a fixed number of steps with a secondary policy [12], or performing many simulated rollouts with each stopping when a terminal state is encountered [10]. Using a single type of planning style is limiting as various situations in an environment might call for a dynamic planning style. For example, when the agent needs to explore the immediate surrounding area a breadth-first search is optimal – instead of proceeding depth-first down one trajectory. If the agent cannot adjust planning styles the resulting plan can be sub-optimal.

In typical planning architectures, the planner is unable efficiently reverse trajectories. The planner must undo previous actions, wasting simulation steps, before continuing down an alternative path. If the planner does not have enough remaining simulation steps, to fully or even partially undo a sub-optimal trajectory, the agent using this plan might perform poor actions. Additionally, if certain actions cannot be reversed the planner might try a nonsensical move, which violates environment dynamics, and produce a plan with an unrealistic prediction. Again, this would lead the agent using this plan to incorrectly value a particular path forward. Ideally, in both of the aforementioned cases, the planner would be able to either "reset" the planning trajectory in one step or could "undo" the last action – bypassing the limitations imposed by the environment and state-transition model.

DPN aims to improve the planning efficiency via various architectural choices. By providing DPN's planner with a triplet of options to "reset", "undo", or "continue" from tracked states during planning it can avoid sub-optimal trajectories and dead-ends. Additionally, as DPN's planner is based on a recurrent network and has no imposed planning structure which, together with the tracked triplet of state, allows it to dynamically adjust planning styles depending on the current context. The contributions of this work are as follows:

- Dynamic Planning Network: a planning architecture that create plans with a learned dynamic planning style.
- We show that providing a planner with the option to choose *where* to plan from improves performance by reducing sub-optimal trajectories.
- A loss for the planner policy that balances between exploration and exploitation during planning.
- DPN outperforms, both in performance and sample ef-

N. Tasfi, M. A. M. Capretz, Dynamic Planning Networks, Proc. of the IEEE 2021 International Joint Conference on Neural Networks, 2021.



(a) Architecture Overview

(b) A Planning Step

Figure 1: a) Network Architecture: Encoder is comprised of several convolutional layers and a fully-connected layer (a box with 3 dots). Planning occurs for $\tau = 1, ..., T$. The result of planning is sent to the agent which emits an action a_t and state-value v_t. Planning uses a fully-connected layer within the agent, outlined in green, to generate an updated hidden state. *b)* A single planning step τ. The planner performs a step of planning using the state-transition model. Circles containing $\times$ indicate multiplication and circles with $\sim$ indicate sampling from the Gumbel Softmax distribution.

ficiency, other planning architectures on commonly used environments in the domain.

The paper is organized as follows: Section II covers our architecture and training procedure, Section III covers related work, Section IV details the experimental design used to evaluate our architecture, and in Section V we analyze the experimental results of our architecture.

II. Dynamic Planning Network

In this section, we describe the design choices of DPN, each architectural component, and training regime used. Steps taken in the environment use subscript t and planning steps use subscript τ.

A. Architectural Choices

DPN is composed of an agent policy π^A, multi-step planner policy $\pi^P_{w,a}$, a state-value function V, a learned state-transition model $\mathcal{M}$, and shared state encoder. Figure 1(a) illustrates a high-level diagram of the DPN architecture.

At its core, DPN extends actor-critic algorithms by adding a pathway dedicated to planning with a learned state-transition model, similar to other planning work [11]–[15]. We define a state-transition model as any model which predicts the next state given an action and the current state.

Using state-transition models in environments with complex dynamics and high dimensional observation spaces has proven difficult as state-transition models must learn from agent experience and require significant amounts of samples and compute [16]–[18]. Often, it is much more efficient to instead learn and make predictions in a lower dimensional space [11]. Therefore, within this work we consider the state-transition model used by Farquhar *et al.* [11] which predicts within the latent embedding space z; where z is produced by an encoder or the state-transition model itself.

Planning components aim to improve the performance of a model-free agent by avoiding costly trial and error in the environment. However the style of planning, the way a state-transition model is used, differs between architectures each with its own benefits. The planning style can be a simple forward rollout [16], [19], enforce a particular structure [11], [20], or use predesigned patterns [12]. In this regard, DPN's architecture is constructed in such a way that it *learns* the best planning pattern to employ. Therefore, DPN's planner is based upon a recurrent neural network which naturally incorporates the recent history in a short-term memory, allowing flexible planning patterns to emerge.

However, an additional issue arises regardless of the planning strategy used: what should the planner do if the last action taken cannot be reversed? Even if, in theory a opposing action exists, there is no guarantee that the state-transition model will produce a coherent prediction. Assuming, temporarily, that the planner chooses an opposing action, which cannot undo the last action but is opposing, and the state-transition model happily follows through: the resulting predicted state would either be nonsensical or unreachable. Therefore to help alleviate this issue, we provide DPN's planner access to an "undo" and "reset" option. This requires tracking a triplet of state embeddings z during planning: the current z^c_τ, previous/undo z^p_τ, and reset/root z^r_τ. Where the root state is the current state of the agent within the environment and the previous state is the last observed state during planning. Both options allow the planner to short-circuit the state-transition model.

Figure 2 illustrates, in a fictional environment during a round of planning, the utility provided by tracking and allowing the planner to select between the triplet of states. From the *top row* of Figure 2, we show how the planner can use a state-transition model and the triplet of states to construct plans. Here, we interpret the plans as the dynamic expansion of a state-action tree. While in the *bottom row* of Figure 2, show the corresponding fictional environment where the red agent must capture the blue goals. The agent can only *push* the grey obstacles which means they can become irreversible stuck as no opposing action exists. The fictional environment illustrates how the added ability to select the "root" or "previous" states gives improved efficiency to the planner.

As the planner progresses through the environment, shown in the *bottom row* of Figure 2a-d, that it pushed the grey obstacle to the left, blocking the goal. By using the "root" state option[1], shown in the *top row* of Figures 2e, the planner can create an alternative route to the goal, shown in the *bottom row* of Figures 2e-f.

If the planner did not have access to the "root" or "previous" states, the resulting trajectory would be sub-optimal in their

[1]The "previous" state option would be a valid choice as well.

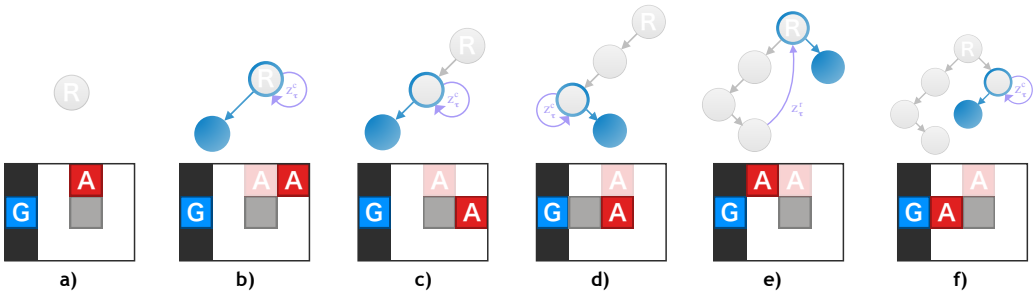


Figure 2: Tree Interpretation. *Top Row*: A tree interpretation of a created plan by DPN. State selections are shown in light purple and state-transitions are shown as blue. The source state is shown as a grey circle with a blue outline and the transitioned state as a fully blue circle. *Bottom Row*: A fictional environment in which the red agent must visit blue goals and can only *push*, and not *pull*, grey obstacles around. The faded agent is meant to signify the current state of the agent in the environment.

predictive value or might contain incorrect information had the state-transition model violated the environment dynamics. Additionally, even if an opposing action did exist, in this case *pull*, the planner would waste planning steps to unroll this poor decision. Planning efficiency becomes especially important when a limited number of planning steps are budgeted.

In DPN, our planner runs for a fixed number of planning steps T, interacting with the state-transition model $\mathcal{M}$, before the agent selects an action a_t in the environment. Pseudo-code is provided for one step of acting by DPN in Algorithm 1. The weight W^A belongs to the agent and its output, given some embedding z, captures the agent's current view of the state in an embedding h^A. We refer to this as the "hidden state" of the agent. This is shown Figure 1(a) as the bottom pathway where W^A is the box with a green border.

At each planning timestep τ where $\tau \in \{1, \dots, T\}$, the planner's policy, in a two-step manner, picks which state of the triplet to plan from using a sampled weighting w^*_τ and then selects an appropriate action a^*_τ given this selected state and history. The weighting w^* and action a^* are sampled using the Gumbel-Softmax trick [21] so we can learn in an end-to-end manner. The planner uses the state-transition model $\mathcal{M}$ to predict the next state $z^*_{\tau+1}$ given the selected state z^*_τ and action a^*_τ. The triplet of embeddings are then updated.

Algorithm 1 Pseudo-code for action selection with DPN

// Given StateModel, Encoder, Planner, and Agent policy.
// Given current state x_t.
$h^P_{t,\tau=0} \leftarrow$ init hidden state of Planner
$z_t = \text{Encoder}(x_t)$
$z^p_{t,\tau=1}, z^c_{t,\tau=1}, z^r_{t,\tau=1} = z_t$
for $\tau \in \{0, \cdots, T-1\}$ **do**
  $h^A_{t,\tau} = W^A z^c_{t,\tau}$
  $h^P_{t,\tau} = \text{RNN}([z^p_{t,\tau}, z^c_{t,\tau}, z^r_{t,\tau}, \frac{\tau}{T}, h^A_{t,\tau}], h^P_{t,\tau-1})$
  $w^*_{t,\tau} = \pi^P_w(\cdot|h^P_{t,\tau})$ // 1-hot action
  $z^*_{t,\tau} = [z^p_{t,\tau}, z^c_{t,\tau}, z^r_{t,\tau}]^T w^*_{t,\tau}$
  $a^*_{t,\tau} = \pi^P_a(\cdot|h^P_{t,\tau}; z^*_{t,\tau})$ // 1-hot action
  $z^*_{t,\tau+1} = \mathcal{M}(a^*_{t,\tau}, z^*_{t,\tau})$
  $z^c_{t,\tau}, z^p_{t,\tau} = z^*_{t,\tau+1}, z^c_{t,\tau}$
end for
$h^A_t = W^A z_t$
$a_t = \pi^A(a|h^A_t, h^P_{t,\tau=T})$

The planner is provided with a context comprised of the current triplet of embeddings, a float indicating the planning step, and the "hidden state" of the agent. Inclusion of the agent's "hidden state" h^A in the planner's is detailed when we discuss training, but briefly: the planner is partially trained to maximize the "surprise" of the agent so providing this information to the planner is beneficial. As the planner uses a recurrent network, we found it best if the agent's policy π^A also consumes the final hidden state $h^P_{\tau=T}$ produced by the planner, shown in Figure 1(a). Doing so forces the planner to keep a running summary of the constructed plan and provides the agent's policy π^A with additional context.

B. Architecture Components

Model-free Pathway: As seen in Figure 1(a), the components along the model-free path, that is the bottom connections, are nearly identical to architectures used by actor-critic methods. Containing an convolutional encoder, optional hidden layers, and two outputs each representing the learned policy and state-value function. In this case the convolutional encoder processes a 2d input image $x_t \in \mathbb{R}^{C \times W \times H}$, with C channels and dimensions $W \times H$, to produce an embedded representation $z_t \in \mathbb{R}^Z$. We refer to the parameters of the actor's policy with θ_a and state-value function as θ_v. As the encoder is shared, it's parameters are a subset of all component parameters in DPN and therefore not explicitly mentioned.

From Figure 1(a), we see an additional two fully-connected layers along with the planning component. The bottom layer, outlined in green, is used to represent the agents current representation of the environment. It is used by the planner to estimate the "surprise" its plans provide. While the fully-connected layer, along the top connecting the planner to the agent, is used to further processes plans such that the agent can learn to extract pertinent information. We found this helpful as the produced plans might contain inaccuracies, caused by repeated application of the state-transition model, or contains non-actionable information. Weber *et al.* [12] use a module in I2A, which they refer to as a rollout encoder, with similar functional and purpose.

Planner: As shown in Figure 1(b), the planner is comprised of a recurrent neural network (RNN) and two fully connected layers. Each layer represents either the sub-policy used to choose a^* or w^*. The weight $w^*_\tau \in [0, 1]^3$ only considers

the current hidden state $h_\tau^P \in \mathbb{R}^H$ produced by the RNN. However, the simulated action $a_\tau^* \in \mathcal{A}$ considers both h_τ^P and the selected embedded state $z_\tau^* \in \mathbb{R}^Z$. We refer to the collective parameters of the planner as θ_p.

Figure 1(b) is primarily used to show the flow of information during one planning step τ. We can clearly see how tightly coupled the interactions between the planner and state-transition model are and how the planner is can fully manipulate the state-transition model based on the selected state z_τ^* and action a_τ^*.

State-Transition Model: The state-transition model is composed of two computation steps with a residual connection in between. The first step is meant to compute an action agnostic representation of the current state embedding z_τ. While the second computes the expected change to the environment given an action a_i. It is defined as follows:

$$\begin{aligned} z' &= z_\tau + tanh(\boldsymbol{W}^{\text{env}} z_\tau^*) \\ z_{\tau+1}^* &= z' + tanh(\boldsymbol{W}^{a_i} z') \end{aligned} \tag{1}$$

both $\boldsymbol{W}^{env} \in \mathbb{R}^{Z \times Z}$ and $\boldsymbol{W}^{a_i} \in \mathbb{R}^{\mathcal{A} \times Z \times Z}$ are learnable parameters of the state-transition model and are referred collectively to as θ_m[2]. We use the same state-transition model presented by Farquhar *et al.* [11] and also perform normalization of z^* after prediction. Doing so keeps the magnitude of the representation more consistent after several application of the state-transition model.

C. Training Details

DPN is trained to maximize the expected reward and as such we train the encoder, value network, and agent's policy using the k-step synchronous version of the advantage actor-critic algorithm (A2C) [2]. We refer to their collective loss, excluding the entropy regularization term, as $\mathcal{L}_{\text{A2C}}$

We treat the planner's policy as an actor, fitting into the A2C framework loss as an additive term, but make two adjustments. First, the planner is trained within planning trajectories only, such that state-transitions in its "environment" are emulated by the state-transition model. This means that for a k-step trajectory, under A2C, the planner will see $k \times T$ samples. Second, the planner's reward is redefined to be the composition between the state-value the agent predicts for the next state $z_{\tau+1}$ and distance between the agent's hidden representations from states z_τ to $z_{\tau+1}$. We term this pseudo-reward as the utility the planner provides to the agent and define it as:

$$\mathcal{U}_\tau(h_{\tau+1}^A, h_\tau^A, z_{\tau+1}) = V(z_{\tau+1}; \theta_v) + \mathbf{D}[h_{\tau+1}^A, h_\tau^A] \tag{2}$$

where $z_{\tau+1}$ is the state transitioned to after performing an action a_τ in state z_τ, h_τ^A and $h_{\tau+1}^A$ are the hidden states of the agent after perceiving the current state z_τ and state transitioned to $z_{\tau+1}$ respectively, $\mathbf{D}$ is a distance measure, and $V(z_{\tau+1})$ is the value the agent assigns the next state $z_{\tau+1}$.

The two terms in Equation 2 tease between exploitation and exploration during planning. If only state-value term $V(.)$, analogous to the reward, where to be maximized by the planner then the produced plans would aim to maximize the reward – exploiting what is already known. In the opposite direction, if the planner focuses only on the distance term $\mathbf{D}$, then it will chose states producing larger differences in the agent's hidden state. More than likely, this would correspond to states which involve some "surprise" to the agent. A similar formulation has been proposed, outside of the planning domain, in work on intrinsic motivation; where the agent sees an external reward r_{ext} and an internal reward r_{int} [22]. This formulation also balances between the notion of exploration and exploitation, as the internally generated reward can be tangential to the reward produced by the environment. We extend this idea to the planning domain.

A secondary choice in the design of utility in Equation 2 is how to combine between the state-value and distance term. We found that an additive distance term helps useful reward signals through, instead of being attenuated, as the reward and distance can disagree on the usefulness of the next state; such as subsequent states with a epsilon distance but non-zero rewards. Here, a multiplicative term can hinder learning as either quantity can be a near-zero number, such as the reward, causing the provided utility to register as essentially zero. Additionally, in the initial stages of our work, we did indeed consider a variant with a multiplicative distance term $\mathbf{D}$ but found sub-par performance when compared to an additive distance term. We hypothesize that the aforementioned effects caused the gradients to vanish, slowing learning along the planning pathway. Following from this, the planner is trained as a policy network only, with the loss $\mathcal{L}_P$ over the planning sequence T:

$$\mathcal{L}_p = \tfrac{1}{T-1} \textstyle\sum_{\tau=0}^{T-1} \nabla_{\theta_p} log \pi_{w,a}^P(\cdot|z_\tau; \theta_p)\, \mathcal{U}_\tau(h_{\tau+1}^A, h_\tau^A, z_{\tau+1}) \tag{3}$$

During parameter updates to the planners parameters we

[2]The action matrix is selected by multiplying a 1-hot encoding of the action.

Algorithm 2 Pseudo-code for DPN

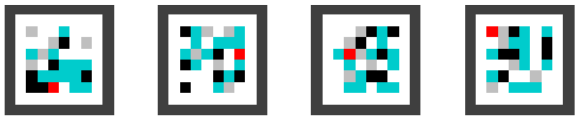
Initialize parameters θ_a, θ_p, θ_v, and θ_m.
repeat
  for $i \in \{0, \cdots, k\}$ **do**
    Pick a_i by calling Algorithm 1 with x_t.
    Receive reward r_i and new state x_{i+1}.
  end for

$$R \leftarrow \begin{cases} 0 & \text{for terminal } x_k \\ V(z_k; \theta_v) & \text{otherwise} \end{cases}$$

  Reset gradients: $d\theta_{\{a,o,v,m\}} \leftarrow 0$.
  for $i \in \{k-1, \cdots, 0\}$ **do**
    for $\tau \in \{0, \cdots, T-1\}$ **do**
      $\mathcal{U}_{i,\tau} = V(z_{i,\tau+1}; \theta_v) + \mathbf{D}[h_{i,\tau+1}^A, h_{i,\tau}^A]$
      $d\theta_p \leftarrow d\theta_p + \nabla_{\theta_p} log \pi_{w,a}^P(\cdot|z_{i,\tau}; \theta_p) \mathcal{U}_{i,\tau}$
    end for

$$R \leftarrow \begin{cases} 0 & \text{for terminal } s_i \\ r_i + \gamma R & \text{otherwise} \end{cases}$$

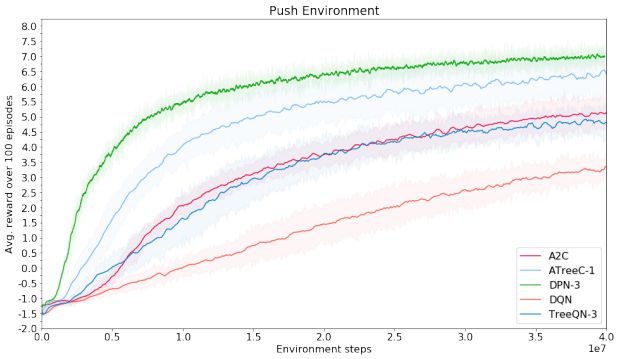
    $d\theta_a \leftarrow d\theta_a + \bar{\rho}_i \nabla_{\theta_a} \log \pi(a_i|s_i; \theta_a)\big\{R - V(z_i; \theta_v)\big\}$
    $d\theta_v \leftarrow d\theta_v + \nabla_{\theta_v} \big\{R - V(z_i; \theta_v)\big\}^2$
    $d\theta_m \leftarrow d\theta_m + \nabla_{\theta_m} \big\{z_{i+1} - \mathcal{M}\big(z_i, a_i; \theta_m\big)\big\}^2$
  end for
  Perform update of θ_p using $d\theta_p$, θ_a with $d\theta_a$, θ_v with $d\theta_v$, and θ_m with $d\theta_m$.
until Max iteration or time reached.



(a) Push Environment Samples.

Model	Avg. Reward
A2C	5.62
ATreeC-1	6.68
DPN-T3	**6.99**
DQN	3.96
TreeQN-3	5.08

(b) Model Performance.



(c) Training Curve.

Figure 3: *Push Environment. a)* Randomly generated samples of the Push environment. Each square's coloring represents a different entity: the agent is shown as red, boxes as aqua, obstacles as black, and goals as grey. The outside of the environment, not visible to the agent, is shown as a black border around the map. *b)* The performance of each model where *Avg. Reward* is the average of the last 1000 episodes of training. *c)* Training curves with DPN compared to various baselines on Push environment.

block gradient computations to the parameters belonging to the agent. In this case the state-value function and the weight W^A used to update the agent's hidden state. We perform updates to the planner in this way as to stop the planner from cheating by modifying the parameters of the agent that define its reward via the quantities in Equation 2. Various choices for distance functions exist, such as the cosine or L2 distance function. In this work we use the L1 distance function, as after empirical evaluation it was the most performant. Results of this evaluation are provided in Section IV.

We train the state-transition model by performing state grounding. As such, it is trained to minimize the L2 distance between the next embedded state z_{t+1}, produced by the encoder, and its prediction $\hat{z}_{t+1}$. The state-transition model makes its prediction from an embedding of the current state z_t and the action taken by the agent a_t that resulted in z_{t+1} [11]:

$$\mathcal{L}_{\mathcal{M}} = \left\{ z_{t+1} - \mathcal{M}(z_t, a_t; \theta_m) \right\}^2 \tag{4}$$

Combining our losses, the architecture is trained using the following gradient:

$$\Delta\theta = \nabla_{\theta_{\text{A2C}}}\mathcal{L}_{\text{A2C}} + \nabla_{\theta_p}\mathcal{L}_P + \lambda\nabla_{\theta_M}\mathcal{L}_{\mathcal{M}} - \beta\nabla_{\theta_{\{a,p\}}}H \tag{5}$$

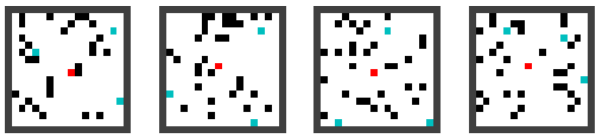
where $\mathcal{L}_{\text{A2C}}$ is the agent's loss, both its policy and value function, $\mathcal{L}_P$ is the planner loss, λ is a hyperparameter controlling the state-grounding loss, H is the entropy regularizer computed for the agent and planner's policies, and β is a hyperparameter tuning entropy maximization of all policies; we used the same β value for each policy. The losses $\mathcal{L}_{\text{A2C}}$ and $\mathcal{L}_{\mathcal{Z}}$ are computed over all parameters; while $\mathcal{L}_P$ and its entropy regularizer losses are computed with respect to only the planner's parameters. DPN is fully specified in Algorithm 2.

III. Related Work

Various efforts have been made to combine model-free and model-based methods, such as the *Dyna-Q* algorithm [23] that learns a model of the environment and uses this model to train a model-free policy. Originally applied in the discrete setting, Gu *et al.* [24] extended Dyna-Q to continuous control. However, none of the aforementioned algorithms use the learned model to improve the online performance of the policy and instead use the model for offline training. Therefore, the learned models are typically trained with a tangential objective to that of the policy such as a high-dimensional reconstruction. In contrast, our work learns a model in an end-to-end manner, such that the model is also optimized for its actual use in planning instead of just prediction.

Pascanu *et al.* [14] implemented a model-based architecture comprised of several individually trained components that learn to construct and execute plans. In contrast, the planning policy used by DPN also selects which state to plan from while Pascanu *et al.* [14] used a separate specialized policy. They examine performance on Gridworld tasks with single and multi-goal variants but on an extremely limited set of small maps. Vezhnevets *et al.* [13] proposed a method which learns to initialize and update a plan; their work does not use a state-transition model and maps new observations to plan updates. Guez *et al.* [25] proposed MCTSnets, an approach for learning to search where they replicate the process used by MCTS. MCTSnets replaces the traditional MCTS components by neural network analogs. The modified procedure evaluates, expands, and back-ups a vector embedding instead of a scalar value. The entire architecture is end-to-end differentiable.

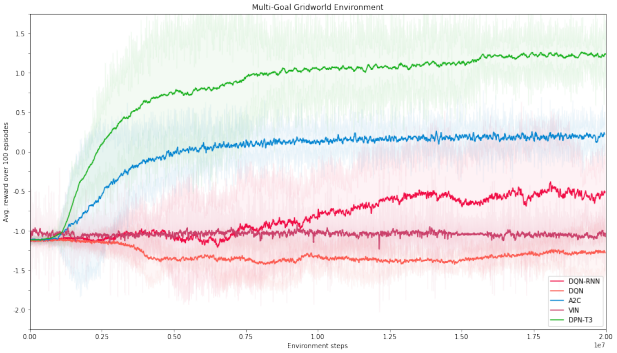
Value prediction networks (VPNs) by Oh *et al.* [19], *Predictron* by Silver *et al.* [26], and *ATreeC* by Farquhar *et al.* [11], an expansion of VPNs, combine learning and planning by training deep networks to plan through iterative rollouts. The *Predictron* predicts values by learning an abstract state-transition function. Oh *et al.* [19] and Farquhar *et al.* [11] both construct trees to improve value estimates by using forward-only rollouts by exhaustively expanding each state's actions. Farquhar *et al.* [11] use the tree for both training and acting. Similarly, Francois-Lavet *et al.* [27] proposed a model that



(a) Multi-Goal Gridworld Environment Samples.

Model	Avg. Reward
DQN-RNN	-0.51
DQN	-1.26
A2C	0.21
VIN	-1.04
DPN-T3	**1.3**

(b) Model Performance.



(c) Training Curve.

Figure 4: *Multi-Goal Gridworld Environment. a)* Randomly generated samples of a 16×16 Multi-Goal Gridworld environments. The agent is shown as red, goals in cyan, obstacles as black, and outside of the environment, not visible to the agent, is shown with a black border. *b)* The performance of each model where *Avg. Reward* is the average of the last 1000 episodes of training. *c)* Training curves with DPN compared to various baselines on 16×16 Gridworld with 3 goals.

combined model-free and model-based components to plan on embedded state representations in a similar fashion to TreeQN [11]. They propose an additional loss to the objective function, an approximate entropy maximization penalty, that ensures the expressiveness of the learned embedding. In contrast to the aforementioned works, during planning DPN learns to selectively expand actions at each state, with the ability to adjust sub-optimal actions, and uses planning results to improve the policy during both training and acting.

Weber *et al.* [12] proposed Imagination Augmented Agents (I2As), an architecture that learns to plan using a separately trained state-transition model. Planning is accomplished by expanding all available actions $\mathcal{A}$ of the initial state and then performing $\mathcal{A}$ rollouts using a tied-policy for a fixed number of steps. In contrast, our work learns the state-transition model end-to-end, uses a separate policy for planning and acting, and is able to dynamically adjust planning rollouts. Empirically, we found that using the same policy for planning and acting caused poor performance. We hypothesize that the optimal policy for planning is inherently different from the one required for optimal control in the environment; as during planning, a bias toward exploration might be optimal.

Tamar *et al.* [28] trained a model-free agent with an explicit differentiable planning structure, implemented with convolutions, to perform approximate on-the-fly value iteration. As their planning structure relies on convolutions, the range of applicable environments is restricted to those where state-transitions can be expressed spatially.

Additional connections between learning environment models, planning and controls, and other methods related to ours were previously discussed by Schmidhuber [29].

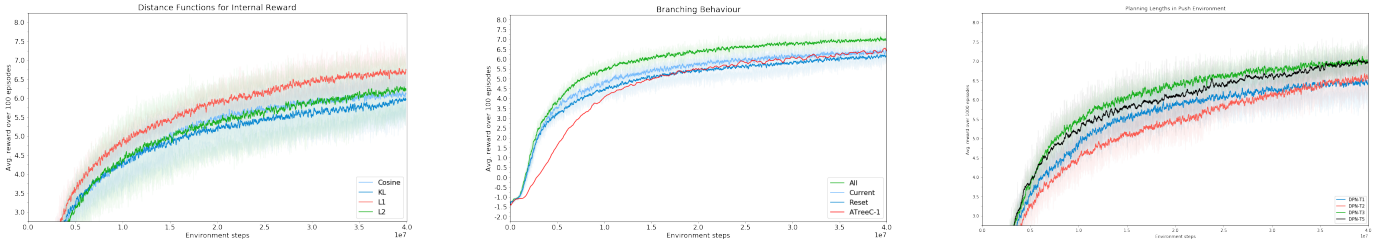
IV. Experiments

We evaluated DPN on a Multi-Goal Gridworld environment and Push [11], a box-pushing puzzle environment. Push is similar to Sokoban used by Weber *et al.* [12] with comparable difficulty. Within our experiments, we evaluated our model performance against either model-free baselines (A2C, DQN, and VIN) [3] or planning baselines (TreeQN and ATreeC). The experiments are designed such that a new scenario is generated across each episode, which ensures that the solution of a single variation cannot be memorized. We are interested in understanding how well our model can adapt to varied scenarios. Additionally, we investigate how planning length T affects model performance, how planner branching affects performance, different distance functions for the planner's reward function, and planning patterns that our agent learned in the Push environment. Full details of the environments, experimental setup, hyperparameters are provided in the supplemental material. Unless specified otherwise, each model configuration is averaged over 3 different seeds and is trained for 40 million steps. As mentioned earlier, we use a version of A2C algorithm with 16 workers, the RMSprop optimizer [30] with a learning rate of $5e-4$ and $\epsilon = 1e-5$.

Push: The Push environment is a box-pushing domain, where an agent must push boxes into goals while avoiding obstacles, with samples shown in Figure 3(a). Since the agent can only push boxes, with no pull actions, poor actions within the environment can lead to irreversible configurations. The agent is randomly placed, along with 12 boxes, 5 goals, and 6 obstacles on the center 6x6 tiles of an 8x8 grid. Boxes cannot be pushed into each other and obstacles are "soft" such that they do not block movement, but generate a negative reward if the agent or a box moves onto an obstacle. Boxes are removed once pushed onto a goal. We use the open-source implementation provided by Farquhar *et al.* [31]. The episode ends when the agent collects all goals, steps off the map, or goes over 75 steps. We compare our model performance against planning baselines, TreeQN and ATreeC [11], as well as model-free baselines, DQN [1] and A2C [2].

Multi-Goal Gridworld: We use a Multi-Goal Gridworld domain with randomly placed obstacles that an agent must navigate searching for goals. The environment, randomly generated

[3] We tested both vanilla implementations and versions using our architecture. A version of DPN with the planning components disabled, equivalent to an A2C model, was evaluated as well. We used the best performing version.



(a) Distance Functions (b) Planner Branching (c) Planning Length

Figure 5: a) *Distance Functions*: the performance of different distance functions. Centered on curve differences. b) *Planner Branching*: Various branching choices for the planner. *All* corresponds to the default architecture, *Current* results in a forward rollout, and *Reset* is the same as 1-step look ahead. ATreeC-1 corresponds to 1-step look ahead as well. c) *Planning Length*: Training over varying planning lengths, $T = \{1, 2, 3, 5\}$, in the Push Environment. Centered on curve differences.

between episodes, is a 16x16 grid with 3 goals. We force a minimum distance between goals and between the agent and goals. The agent must learn an optimal policy to solve new unseen maps. Figure 4(a) shows several instances of a 16x16 Multi-Goal Gridworld. The rewards that an agent receives are as follows: +1 for each goal captured, -1 for colliding with a wall, -1 for stepping off the map, -0.01 for each step, and -1 for going over the step limit. An episode terminates if the agent collides with an obstacle, collects all the goals, steps off the map, or goes over 70 steps. We evaluate our algorithm against model-free baselines such as A2C [2], variants of DQN (recurrent and non-recurrent) [1], and Value Iteration Network (VIN) [28]. Each baseline, with the exception of VIN, used the same encoder structure as DPN. We train for 20 million environment steps.

Planner Distance Functions: We vary the distance function used by the planner's loss defined in Equation 2. We examine L1, L2, KL, and Cosine distance functions.

Planner Branching: We examine the affect on performance of different branching options: current, reset, or all. We also included the ATreeC-1 baseline as this corresponds to the reset branching option of our architecture and serves as a sanity check.

Planning Length: Using the Push environment, we varied the parameter T, which adjusts the number of planning steps, with $T = \{1, 2, 3, 5\}$ evaluated. The Push environment was chosen because the performance is sensitive to an agent's ability to plan effectively.

Planning Patterns: We examine the planning patterns that our agent learns in the Push environment with $T = 5$. Here we are interested in understanding what information the agent extracts from the simulation as context before acting.

V. Results and Discussion

A. Push Environment

Figure 3(c) shows DPN compared to DQN, A2C, TreeQN and ATreeC baselines [4]. For TreeQN and ATreeC, we chose tree depths which gave the best performance, corresponding to tree depths of 3 and 1 respectively. Our model clearly outperforms both planning and non-planning baselines: TreeQN, ATreeC, DQN, and A2C. We see that our architecture converges at a faster rate than the other baselines, matching ATreeC-1's final performance after roughly 20 million steps in the environment. In comparison to the other planning baselines, TreeQN and ATreeC, require roughly 35-40 million steps: $\sim$2x additional samples.

We note that the planning efficiency of DPN is higher in terms of overall performance per number of state-transitions. On the Push environment, with $\mathcal{A} = 4$ actions, TreeQN with tree depth of $d = 3$ requires $\left(\frac{\mathcal{A}^{d+1}-1}{\mathcal{A}-1}\right) - 1 = 84$ state-transitions. In contrast, using DPN with a planning length of $T = 3$ requires only T state-transitions – a 96% reduction. Loosely comparing to I2As, simply in terms of state-transitions, we see that I2As require $\mathcal{A} \times L$ state-transitions per action step, where L is the rollout length. This performance improvement is a result of DPN learning to selectively expand actions and being able to dynamically adjust previously simulated actions during planning.

B. Multi-Goal Gridworld

Figure 4(c) shows, the results of DPN compared to various model-free baselines. Within this domain, the difference in performance is clear: our model outperforms the baselines by a significant margin. The policies that DPN learns generalizes better to new scenarios, can effectively avoid obstacles, and is able to capture multiple goals. Of the model-free baselines, we see that the A2C baseline performs the best. We believe that the A2C baseline is able to better explore the environment due to the multiple workers running in parallel throughout training. We trained VIN without curriculum learning for 20million steps with near identical settings[5] prescribed by Tamar *et al.* [28]. As seen in Figure 4(c), the DQN variants and VIN fail to capture any goals and do not achieve a score higher than -1.0. It should be noted we saw little performance improvement even when allocating the A2C and DQN baselines an additional 2x environment steps (40 million) or, in the case of DQN models, a 2-4x longer exploration period (8-16 million). The

[4] The data for the training curves of DQN, A2C, TreeQN, and ATreeC were provided by Farquhar *et al.* via email correspondence. Each experiment was run with 12 different seeds for 40 million steps.

[5] The original results with VIN relied on curriculum learning. To provide a fair comparison all methods are trained without curriculum learning.

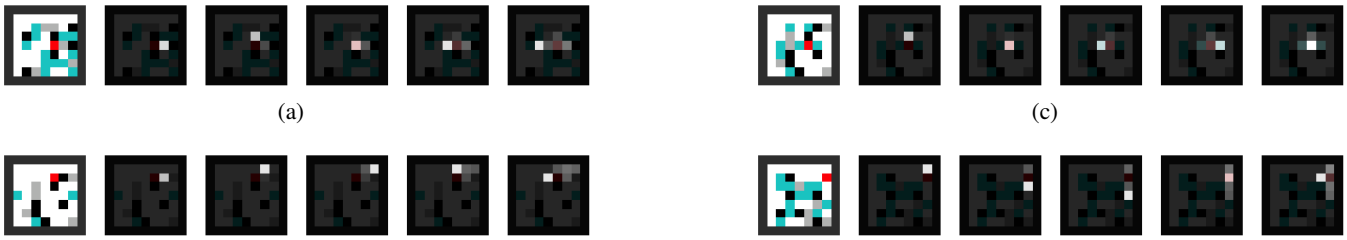


Figure 6: Samples of observed planning-type patterns the agent uses in the Push environment with $T = 5$. The faded environments, to the right of each sample, is used to signify when the agent is planning. Highlighted squares represent the location that the planner chose to move towards during planning. Faded squares show where it has been during planning.

poor performance of the baseline models might be the result of high variance in the environment's configuration between episodes and sparse rewards. We believe that DPN performs better because it captures common structure present between all permutations of the environment by using the environment model. This allows it to exploit the model for planning in newly generated mazes.

C. Planner Distance Functions

From 5(a), we see an evaluation of distance functions used in Equation 2. The L1 distance function has the best performance with slightly faster convergence. While the L2, Cosine, and KL functions have worse performance. We hypothesize that the L1 distance performed better due to its robustness to outliers, a likely event during learning, as the distance is a function of noisy and changing vectors from the agent and state-transition model.

D. Planner Branching

In Figure 5(b), we see how different branching options affects the architecture performance. Our proposed branching improves performance of the architecture as compared to the *current* and *reset* options. Interestingly, the performance of our architecture when using the *reset* options is roughly the same as ATreeC-1. This is unsurprising as the *reset* and ATreeC-1 options employ a similar planning strategy of a shallow 1-step look-ahead. The small discrepancy in performance could be due to ATreeC-1 evaluating all 4 actions while DPN evaluates only 3. We see that the *current* branching option results in better performance and amounts to a forward-only rollout. We hypothesize the performance difference between *current* and *reset* is from DPN being able to see the results of its actions from the first planning step over a longer time span.

E. Planning Length

In Figure 5(c), we see the performance of our model over the planning lengths $T = \{1, 2, 3, 5\}$. As seen in Figure 5(c), model performance increases as we add additional planning steps, while the number of model parameters remains constant.

As the planning length increases, we see the general trend of faster model convergence. Even a single step $T = 1$ of planning allows the agent to test action-hypotheses and avoid poor choices in the environment. From Figure 5(c) we see that an additional planning step, from $T = 1$ to $T = 2$, does not provide benefit until later in training. We see that both the planning lengths $T = 3$ and $T = 5$ tie for the best performance. Similar to Farquhar *et al.* [11], we hypothesize that this is due to a ceiling effect in this domain. This is clear as there are diminishing returns in performance with increased planning lengths. In terms of the shorter planning lengths, we suspect that they do not allow the planner to learn a policy that provides enough utility to the agent. Ideally, the architecture would be able to adjust the number of planning steps T dynamically based on current needs. We see this expansion, similar to the adaptive computation presented by Graves [32], as an interesting avenue for future work.

F. Planning Patterns

By watching a trained agent play through newly generated maps, we observe the possible emergence of planning-type patterns, which are shown in Figure 6 for $T = 5$. We see what appears to be a mixture between breadth-first search (BFS) and depth-first search (DFS). We found the resulting patterns to be quite consistent. Furthermore, the entropy of the planner's policy was quite low at the end of training, which indicates the produces planning patterns are near-deterministic. In Figure 6(a) we see the agent does first performs a partial breadth-first search around itself before continuing farther left. Figure 6(b) shows the agent learning to exploit the "previous" state, which is a "lazy reset" to explore the upper right corner. The agent moves as follows: right with the current state, upward with the current state, right again with the current state, left from the previous state, and right one last time after resetting. In Figure 6(c) we see the planner does a partial breadth first search around the agent. In this case the planner used the current state for the entire planning trajectory. Finally, in Figure 6(d) we see the planner alternate between a breadth-first and depth-first search. The planner moves upward from the root state, resets moving downward, picks the current state moving downward once more, selects the previous state while moving up, and then finishes by moving left using the current state.

VI. Conclusion

In this paper, we have presented DPN, a new architecture for deep reinforcement learning that uses a planner and agent working in tandem. The planner is optimized to maximize a pseudo-reward, the utility provided to the agent, which balanced between exploitation and exploration during planning. We have demonstrated that DPN outperforms the model-free and planning baselines in both the Mutli-Goal Gridworld and Push environments while using $\sim$2x fewer environment samples. Furthermore, the ability for DPN to learn a dynamic planning style enables it to achieve much greater efficiency in terms of the state-transitions required; this is especially evident when comparing between TreeQN, with a fixed planning style, and DPN, with a dynamic planning style. By letting the planner learn it's own planning style we see evidence of emergent planning patterns appear, such as breadth-first search. In the Push environment we see DPN achieves greater or equal performance to TreeQN while requiring 96% percent fewer applications of the state-transition model. Taken all together, DPN, in comparison to other architectures, reduces the computational requirements to reach a similar level of performance. Finally, we have shown that giving the planner the option to select *where* to plan from helps avoid sub-optimal trajectories. In our studies we have provided evidence that the triplet previous, current, reset provides the greatest performance. In future work we plan to examine how structured memory can help improve this dynamic planning process.

VII. Acknowledgments

We would like to thank Eder Santana, Tony Zhang, and Justin Tomasi for helpful feedback and discussion. This project received funding from Ontario Centres of Excellence, Voucher for Innovation and Productivity (VIPI) Program Project #29393.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[2] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International conference on machine learning*, 2016, pp. 1928–1937.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[4] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[5] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, second edition, in progress ed. MIT press, 2017.

[6] J. Peng and R. J. Williams, "Efficient learning and planning within the dyna framework," *Adaptive Behavior*, vol. 1, no. 4, pp. 437–454, 1993.

[7] M. Deisenroth and C. 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)*, 2011, pp. 465–472.

[8] S. Levine and P. Abbeel, "Learning neural network policies with guided policy search under unknown dynamics," in *Advances in Neural Information Processing Systems*, 2014, pp. 1071–1079.

[9] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *International conference on computers and games*. Springer, 2006, pp. 72–83.

[10] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[11] G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson, "Treeqn and atreec: Differentiable tree planning for deep reinforcement learning," *arXiv preprint arXiv:1710.11417*, 2017.

[12] T. Weber, S. Racanière, D. P. Reichert, L. Buesing, A. Guez, D. J. Rezende, A. P. Badia, O. Vinyals, N. Heess, Y. Li *et al.*, "Imagination-augmented agents for deep reinforcement learning," *arXiv preprint arXiv:1707.06203*, 2017.

[13] A. Vezhnevets, V. Mnih, J. Agapiou, S. Osindero, A. Graves, O. Vinyals, K. Kavukcuoglu *et al.*, "Strategic attentive writer for learning macro-actions," *arXiv preprint arXiv:1606.04695*, 2016.

[14] R. Pascanu, Y. Li, O. Vinyals, N. Heess, L. Buesing, S. Racanière, D. Reichert, T. Weber, D. Wierstra, and P. Battaglia, "Learning model-based planning from scratch," *arXiv preprint arXiv:1707.06170*, 2017.

[15] A. Deac, P. Veličković, O. Milinković, P.-L. Bacon, J. Tang, and M. Nikolić, "Xlvin: executed latent value iteration nets," *arXiv preprint arXiv:2010.13146*, 2020.

[16] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. P. Singh, "Action-conditional video prediction using deep networks in atari games," *CoRR*, vol. abs/1507.08750, 2015. [Online]. Available: http://arxiv.org/abs/1507.08750

[17] S. Chiappa, S. Racaniere, D. Wierstra, and S. Mohamed, "Recurrent environment simulators," *arXiv preprint arXiv:1704.02254*, 2017.

[18] M. Guzdial, B. Li, and M. O. Riedl, "Game engine learning from video," 2017.

[19] J. Oh, S. Singh, and H. Lee, "Value prediction network," *CoRR*, vol. abs/1707.03497, 2017. [Online]. Available: http://arxiv.org/abs/1707.03497

[20] J.-B. Grill, F. Altché, Y. Tang, T. Hubert, M. Valko, I. Antonoglou, and R. Munos, "Monte-Carlo tree search as regularized policy optimization," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 3769–3778. [Online]. Available: http://proceedings.mlr.press/v119/grill20a.html

[21] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *arXiv preprint arXiv:1611.01144*, 2016.

[22] N. Chentanez, A. G. Barto, and S. P. Singh, "Intrinsically motivated reinforcement learning," in *Advances in neural information processing systems*, 2005, pp. 1281–1288.

[23] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[24] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, "Continuous deep q-learning with model-based acceleration," in *International Conference on Machine Learning*, 2016, pp. 2829–2838.

[25] A. Guez, T. Weber, I. Antonoglou, K. Simonyan, O. Vinyals, D. Wierstra, R. Munos, and D. Silver, "Learning to search with mctsnets," *arXiv preprint arXiv:1802.04697*, 2018.

[26] D. Silver, H. van Hasselt, M. Hessel, T. Schaul, A. Guez, T. Harley, G. Dulac-Arnold, D. Reichert, N. Rabinowitz, A. Barreto *et al.*, "The predictron: End-to-end learning and planning," *arXiv preprint arXiv:1612.08810*, 2016.

[27] V. François-Lavet, Y. Bengio, D. Precup, and J. Pineau, "Combined reinforcement learning via abstract representations," *arXiv preprint arXiv:1809.04506*, 2018.

[28] A. Tamar, S. Levine, P. Abbeel, Y. WU, and G. Thomas, "Value iteration networks," in *Advances in Neural Information Processing Systems*, 2016, pp. 2146–2154.

[29] J. Schmidhuber, "On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models," *arXiv preprint arXiv:1511.09249*, 2015.

[30] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[31] G. Farquhar, T. Rocktäschel, M. Igl, and S. Whiteson, "https://github.com/oxwhirl/treeqn/blob/master/treeqn/envs/push.py", 2017.

[32] A. Graves, "Adaptive computation time for recurrent neural networks," *arXiv preprint arXiv:1603.08983*, 2016.