The Predictron: End-To-End Learning and Planning

David Silver [*1] **Hado van Hasselt** [*1] **Matteo Hessel** [*1] **Tom Schaul** [*1] **Arthur Guez** [*1] **Tim Harley** [1]
Gabriel Dulac-Arnold [1] **David Reichert** [1] **Neil Rabinowitz** [1] **Andre Barreto** [1] **Thomas Degris** [1]

Abstract

One of the key challenges of artificial intelligence is to learn models that are effective in the context of planning. In this document we introduce the *predictron* architecture. The predictron consists of a fully abstract model, represented by a Markov reward process, that can be rolled forward multiple "imagined" planning steps. Each forward pass of the predictron accumulates internal rewards and values over multiple planning depths. The predictron is trained end-to-end so as to make these accumulated values accurately approximate the true value function. We applied the predictron to procedurally generated random mazes and a simulator for the game of pool. The predictron yielded significantly more accurate predictions than conventional deep neural network architectures.

1. Introduction

The central idea of model-based reinforcement learning is to decompose the RL problem into two subproblems: learning a model of the environment, and then planning with this model. The model is typically represented by a Markov reward process (MRP) or decision process (MDP). The planning component uses this model to evaluate and select among possible strategies. This is typically achieved by rolling forward the model to construct a value function that estimates cumulative reward. In prior work, the model is trained essentially independently of its use within the planner. As a result, the model is not well-matched with the overall objective of the agent. Prior deep reinforcement learning methods have successfully constructed models that can unroll near pixel-perfect reconstructions (Oh et al., 2015; Chiappa et al., 2016); but are yet to surpass state-of-the-art model-free methods in challenging RL domains with raw inputs (e.g., Mnih et al., 2015; 2016; Lillicrap et al., 2016).

In this paper we introduce a new architecture, which we call the *predictron*, that integrates learning and planning into one end-to-end training procedure. At every step, a model is applied to an internal state, to produce a next state, reward, discount, and value estimate. This model is completely abstract and its only goal is to facilitate accurate value prediction. For example, to plan effectively in a game, an agent must be able to predict the score. If our model makes accurate predictions, then an optimal plan with respect to our model will also be optimal for the underlying game – even if the model uses a different state space (e.g., abstract representations of enemy positions, ignoring their shapes and colours), action space (e.g., high-level actions to move away from an enemy), rewards (e.g., a single abstract step could have a higher value than any real reward), or even time-step (e.g., a single abstract step could "jump" the agent to the end of a corridor). All we require is that trajectories through the abstract model produce scores that are consistent with trajectories through the real environment. This is achieved by training the predictron end-to-end, so as to make its value estimates as accurate as possible.

An ideal model could generalise to many different prediction tasks, rather than overfitting to a single task; and could learn from a rich variety of feedback signals, not just a single extrinsic reward. We therefore train the predictron to predict a host of different value functions for a variety of pseudo-reward functions and discount factors. These pseudo-rewards can encode any event or aspect of the environment that the agent may care about, e.g., staying alive or reaching the next room.

We focus upon the prediction task: estimating value functions in MRP environments with uncontrolled dynamics. In this case, the predictron can be implemented as a deep neural network with an MRP as a recurrent core. The predictron unrolls this core multiple steps and accumulates rewards into an overall estimate of value.

We applied the predictron to procedurally generated ran-

[*]Equal contribution [1]DeepMind, London. Correspondence to: David Silver <davidsilver@google.com>, Hado van Hasselt <hado@google.com>, Matteo Hessel <mthss@google.com>, Tom Schaul <schaul@google.com>, Arthur Guez <aguez@google.com>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

dom mazes, and a simulated *pool* domain, directly from pixel inputs. In both cases, the predictron significantly outperformed model-free algorithms with conventional deep network architectures; and was much more robust to architectural choices such as depth.

2. Background

We consider environments defined by an MRP with states $s \in \mathcal{S}$. The MRP is defined by a function, $s', r, \gamma = p(s, \alpha)$, where s' is the next state, r is the reward, and γ is the discount factor, which can for instance represent the non-termination probability for this transition. The process may be stochastic, given IID noise α.

The *return* of an MRP is the cumulative discounted reward over a single trajectory, $g_t = r_{t+1} + \gamma_{t+1} r_{t+2} + \gamma_{t+1}\gamma_{t+2} r_{t+3} + \ldots$, where γ_t can vary per time-step. We consider a generalisation of the MRP setting that includes vector-valued rewards $\mathbf{r}$, diagonal-matrix discounts $\boldsymbol{\gamma}$, and vector-valued returns $\mathbf{g}$; definitions are otherwise identical to the above. We use this bold font notation to closely match the more familiar scalar MRP case; the majority of the paper can be comfortably understood by reading all rewards as scalars, and all discount factors as scalar and constant, i.e., $\gamma_t = \gamma$.

The *value function* of an MRP p is the expected return from state s, $v_p(s) = \mathbb{E}_p\left[\mathbf{g}_t \mid s_t = s\right]$. In the vector case, these are known as *general* value functions (Sutton et al., 2011). We will say that a (general) value function $v(\cdot)$ is *consistent* with environment p if and only if $v = v_p$ which satisfies the following *Bellman equation* (Bellman, 1957),

$$v_p(s) = \mathbb{E}_p\left[\mathbf{r} + \boldsymbol{\gamma} v_p(s') \mid s\right] . \tag{1}$$

In model-based reinforcement learning (Sutton & Barto, 1998), an approximation $m \approx p$ to the environment is learned. In the uncontrolled setting this model is normally an MRP $s', \mathbf{r}, \boldsymbol{\gamma} = m(s, \beta)$ that maps from state s to subsequent state s' and additionally outputs rewards $\mathbf{r}$ and discounts $\boldsymbol{\gamma}$; the model may be stochastic given an IID source of noise β. A (general) value function $v_m(\cdot)$ is consistent with model m (or *valid*, (Sutton, 1995)), if and only if it satisfies a Bellman equation $v_m(s) = \mathbb{E}_m\left[\mathbf{r} + \boldsymbol{\gamma} v_m(s') \mid s\right]$ with respect to model m. Conventionally, model-based RL methods focus on finding a value function v that is consistent with a separately learned model m.

3. Predictron architecture

The predictron is composed of four main components. First, a state representation $\mathbf{s} = f(s)$ that encodes raw input s (this could be a history of observations, in partially observed settings, for example when f is a recurrent network) into an internal (abstract, hidden) state $\mathbf{s}$. Second, a model $\mathbf{s}', \mathbf{r}, \boldsymbol{\gamma} = m(\mathbf{s}, \beta)$ that maps from internal state $\mathbf{s}$ to subsequent internal state $\mathbf{s}'$, internal rewards $\mathbf{r}$, and internal discounts $\boldsymbol{\gamma}$. Third, a value function v that outputs internal values $\mathbf{v} = v(\mathbf{s})$ representing the remaining internal return from internal state $\mathbf{s}$ onwards. The predictron is applied by unrolling its model m multiple "planning" steps to produce internal rewards, discounts and values. We use superscripts $\bullet^k$ to indicate internal steps of the model (which have no necessary connection to time steps $\bullet_t$ of the environment). Finally, these internal rewards, discounts and values are combined together by an *accumulator* into an overall estimate of value $\mathbf{g}$. The whole predictron, from input state s to output, may be viewed as a value function approximator for external targets (i.e., the returns in the real environment). We consider both k-step and λ-weighted accumulators.

The *k-step predictron* rolls its internal model forward k steps (Figure 1a). The 0-step *predictron return* (henceforth abbreviated as *preturn*) is simply the first value $\mathbf{g}^0 = \mathbf{v}^0$, the 1-step preturn is $\mathbf{g}^1 = \mathbf{r}^1 + \boldsymbol{\gamma}^1 \mathbf{v}^1$. More generally, the k-step *predictron return* $\mathbf{g}^k$ is the internal return obtained by accumulating k model steps, plus a discounted final value $\mathbf{v}^k$ from the kth step:

$$\mathbf{g}^k = \mathbf{r}^1 + \boldsymbol{\gamma}^1(\mathbf{r}^2 + \boldsymbol{\gamma}^2(\ldots + \boldsymbol{\gamma}^{k-1}(\mathbf{r}^k + \boldsymbol{\gamma}^k \mathbf{v}^k)\ldots))$$

The *λ-predictron* combines together many k-step preturns. Specifically, it computes a diagonal weight matrix $\boldsymbol{\lambda}^k$ from each internal state s^k. The accumulator uses weights $\boldsymbol{\lambda}^0, \ldots, \boldsymbol{\lambda}^K$ to aggregate over k-step preturns $\mathbf{g}^0, \ldots, \mathbf{g}^K$ and output a combined value that we call the *λ-preturn* $\mathbf{g}^\lambda$,

$$\mathbf{g}^\lambda = \sum_{k=0}^{K} \boldsymbol{w}^k \mathbf{g}^k \tag{2}$$

$$\boldsymbol{w}^k = \begin{cases} (\mathbf{1} - \boldsymbol{\lambda}^k) \prod_{j=0}^{k-1} \boldsymbol{\lambda}^j & \text{if } k < K \\ \prod_{j=0}^{K-1} \boldsymbol{\lambda}^j & \text{otherwise.} \end{cases} \tag{3}$$

where $\mathbf{1}$ is the identity matrix. This λ-preturn is analogous to the λ-return in the forward-view TD(λ) algorithm (Sutton, 1988; Sutton & Barto, 1998). It may also be computed by a backward accumulation through intermediate steps $\mathbf{g}^{k,\lambda}$,

$$\mathbf{g}^{k,\lambda} = (\mathbf{1} - \boldsymbol{\lambda}^k)\mathbf{v}^k + \boldsymbol{\lambda}^k\left(\mathbf{r}^{k+1} + \boldsymbol{\gamma}^{k+1}\mathbf{g}^{k+1,\lambda}\right) , \tag{4}$$

where $\mathbf{g}^{K,\lambda} = \mathbf{v}^K$, and then using $\mathbf{g}^\lambda = \mathbf{g}^{0,\lambda}$. Computation in the λ-predictron operates in a sweep, iterating first through the model from $k = 0 \ldots K$ and then back through the accumulator from $k = K \ldots 0$ in a single "forward" pass of the network (see Figure 1b). Each $\boldsymbol{\lambda}^k$ weight

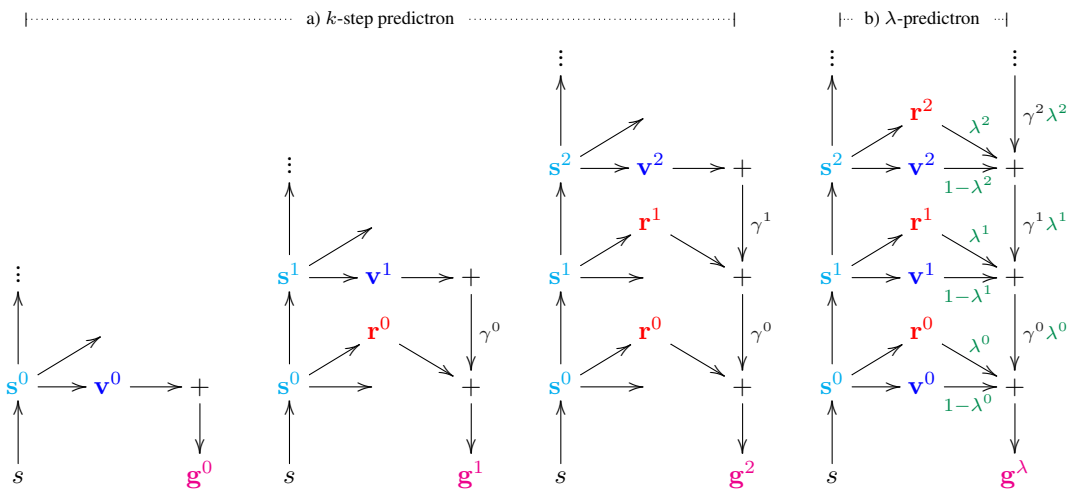

Figure 1. a) The k-step predictron architecture. The first three columns illustrate 0, 1 and 2-step pathways through the predictron. The 0-step preturn reduces to standard model-free value function approximation; other preturns "imagine" additional steps with an internal model. Each pathway outputs a k-step preturn $\mathbf{g}^k$ that accumulates discounted rewards along with a final value estimate. In practice all k-step preturns are computed in a single forward pass. b) The λ-predictron architecture. The λ-parameters gate between the different preturns. The output is a λ-preturn $\mathbf{g}^\lambda$ that is a mixture over the k-step preturns. For example, if $\boldsymbol{\lambda}^0 = \mathbf{1}, \boldsymbol{\lambda}^1 = \mathbf{1}, \boldsymbol{\lambda}^2 = \mathbf{0}$ then we recover the 2-step preturn, $\mathbf{g}^\lambda = \mathbf{g}^2$. Discount factors $\boldsymbol{\gamma}^k$ and λ-parameters $\boldsymbol{\lambda}^k$ are dependent on state $\mathbf{s}^k$; this dependence is not shown in the figure.

acts as a gate on the computation of the λ-preturn: a value of $\boldsymbol{\lambda}^k = \mathbf{0}$ will truncate the λ-preturn at layer k, while a value of $\boldsymbol{\lambda}^k = \mathbf{1}$ will utilise deeper layers based on additional steps of the model m; the final weight is always $\boldsymbol{\lambda}^K = \mathbf{0}$. The individual $\boldsymbol{\lambda}^k$ weights may depend on the corresponding abstract state $\mathbf{s}^k$ and can differ per prediction. This enables the predictron to compute to an adaptive depth (Graves, 2016) depending on the internal state and learning dynamics of the network.

4. Predictron learning updates

We first consider updates that optimise the joint parameters $\boldsymbol{\theta}$ of the state representation, model, and value function. We begin with the k-step predictron. We update the k-step preturn $\mathbf{g}^k$ towards a target outcome $\mathbf{g}$, e.g. the Monte-Carlo return from the real environment, by minimising a mean-squared error loss,

$$L^k = \frac{1}{2} \left\| \mathbb{E}_p \left[\mathbf{g} \mid s \right] - \mathbb{E}_m \left[\mathbf{g}^k \mid s \right] \right\|^2 .$$
$$\frac{\partial l^k}{\partial \boldsymbol{\theta}} = \left(\mathbf{g} - \mathbf{g}^k \right) \frac{\partial \mathbf{g}^k}{\partial \boldsymbol{\theta}} . \tag{5}$$

where $l^k = \frac{1}{2} \left\| \mathbf{g} - \mathbf{g}^k \right\|^2$ is the sample loss. We can use the gradient of the sample loss to update parameters, e.g., by stochastic gradient descent. For stochastic models, independent samples of $\mathbf{g}^k$ and $\frac{\partial \mathbf{g}^k}{\partial \boldsymbol{\theta}}$ are required for unbiased samples of the gradient of L^k.

The λ-predictron combines many k-step preturns. To update the joint parameters $\boldsymbol{\theta}$, we can uniformly average the losses on the individual preturns $\mathbf{g}^k$,

$$L^{0:K} = \frac{1}{2K} \sum_{k=0}^{K} \left\| \mathbb{E}_p \left[\mathbf{g} \mid s \right] - \mathbb{E}_m \left[\mathbf{g}^k \mid s \right] \right\|^2 ,$$
$$\frac{\partial l^{0:K}}{\partial \boldsymbol{\theta}} = \frac{1}{K} \sum_{k=0}^{K} \left(\mathbf{g} - \mathbf{g}^k \right) \frac{\partial \mathbf{g}^k}{\partial \boldsymbol{\theta}} . \tag{6}$$

Alternatively, we could weight each loss by the usage $\boldsymbol{w}^k$ of the corresponding preturn, such that the gradient is $\sum_{k=0}^{K} \boldsymbol{w}^k \left(\mathbf{g} - \mathbf{g}^k \right) \frac{\partial \mathbf{g}^k}{\partial \boldsymbol{\theta}}$.

In the λ-predictron, the $\boldsymbol{\lambda}^k$ weights (that determine the relative weighting $\boldsymbol{w}^k$ of the k-step preturns) depend on additional parameters $\boldsymbol{\eta}$, which are updated so as to minimise a mean-squared error loss L^λ,

$$L^\lambda = \frac{1}{2} \left\| \mathbb{E}_p \left[\mathbf{g} \mid s \right] - \mathbb{E}_m \left[\mathbf{g}^\lambda \mid s \right] \right\|^2 .$$
$$\frac{\partial l^\lambda}{\partial \boldsymbol{\eta}} = \left(\mathbf{g} - \mathbf{g}^\lambda \right) \frac{\partial \mathbf{g}^\lambda}{\partial \boldsymbol{\eta}} . \tag{7}$$

In summary, the joint parameters $\boldsymbol{\theta}$ of the state representation f, the model m, and the value function v are updated to make each of the k-step preturns $\mathbf{g}^k$ more similar to the target $\mathbf{g}$, and the parameters $\boldsymbol{\eta}$ of the λ-accumulator are updated to learn the weights $\boldsymbol{w}^k$ so that the aggregate λ-preturn $\mathbf{g}^\lambda$ becomes more similar to the target $\mathbf{g}$.

4.1. Consistency updates

In model-based reinforcement learning architectures such as Dyna (Sutton, 1990), value functions may be updated using both real and imagined trajectories. The refinement of value estimates based on these imagined trajectories is often referred to as *planning*. A similar opportunity arises in the context of the predictron. Each rollout of the predictron generates a trajectory in abstract space, alongside with rewards, discounts and values. Furthermore, the predictron aggregates these components in multiple value estimates $(\mathbf{g}^0, ..., \mathbf{g}^k, \mathbf{g}^\lambda)$.

We may therefore update each individual value estimate towards the best aggregated estimate. This corresponds to adjusting each preturn $\mathbf{g}^k$ towards the λ-preturn $\mathbf{g}^\lambda$, by minimizing:

$$L = \frac{1}{2}\sum_{k=0}^{K} \left\| \mathbb{E}_m \left[\mathbf{g}^\lambda \mid s \right] - \mathbb{E}_m \left[\mathbf{g}^k \mid s \right] \right\|^2 .$$

$$\frac{\partial l}{\partial \boldsymbol{\theta}} = \sum_{k=0}^{K} \left(\mathbf{g}^\lambda - \mathbf{g}^k \right) \frac{\partial \mathbf{g}^k}{\partial \boldsymbol{\theta}} . \tag{8}$$

Here $\mathbf{g}^\lambda$ is considered fixed; the parameters $\boldsymbol{\theta}$ are only updated to make $\mathbf{g}^k$ more similar to $\mathbf{g}^\lambda$, not vice versa.

These consistency updates do not require any labels $\mathbf{g}$ or samples from the environment. As a result, it can be applied to (potentially hypothetical) states that have no associated 'real' (e.g. Monte-Carlo) outcome: we update the value estimates to be self-consistent with each other. This is especially relevant in the semi-supervised setting, where these consistency updates allow us to exploit the unlabelled inputs.

5. Experiments

We conducted experiments in two domains. The first domain consists of randomly generated mazes. Each location either is empty or contains a wall. In these mazes, we considered two tasks. In the first task, the input was a 13×13 maze and a random initial position and the goal is to predict a trajectory generated by a simple fixed deterministic policy. The target $\mathbf{g}$ was a vector with an element for each cell of the maze which is either one, if that cell was reached by the policy, or zero. In the second random-maze task the goal was to predict for each of the cells on the diagonal of a 20×20 maze (top-left to bottom-right) whether it is connected to the bottom-right corner. Two locations in a maze are considered connected if they are both empty and we can reach one from the other by moving horizontally or vertically through adjacent empty cells. In both cases some predictions would seem to be easier if we could learn a simple algorithm, such as some form of search or flood fill; our hypothesis is that an internal model can learn to emulate such algorithms, where naive approximation may struggle. A few example mazes are shown in Figure 2.

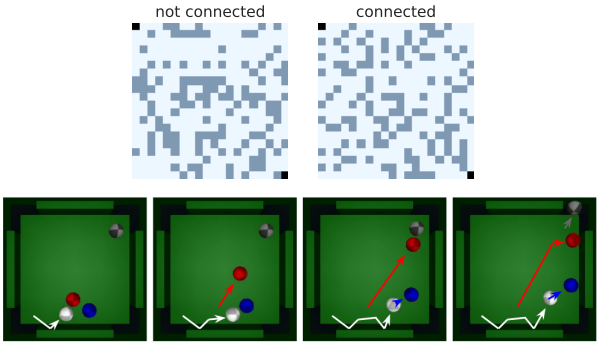

Figure 2. **Top:** Two sample mazes from the random-maze domain. Light blue cells are empty, darker blue cells contain a wall. One maze is connected from top-left to bottom-right, the other is not. **Bottom:** An example trajectory in the pool domain (before downsampling), selected by maximising the prediction by a predictron of pocketing balls.

Our second domain is a simulation of the game of pool, using four balls and four pockets. The simulator is implemented in the physics engine *Mujoco* (Todorov et al., 2012). We generate sequences of RGB frames starting from a random arrangement of balls on the table. The goal is to simultaneously learn to predict future events for each of the four balls, given 5 RGB frames as input. These events include: collision with any other ball, collision with any boundary of the table, entering a quadrant ($\times 4$, for each quadrant), being located in a quadrant ($\times 4$, for each quadrant), and entering a pocket ($\times 4$, for each pocket). Each of these 14×4 events provides a binary pseudo-reward that we combine with 5 different discount factors $\{0, 0.5, 0.9, 0.98, 1\}$ and predict their cumulative discounted sum over various time spans. This yields a total of 280 general value functions. An example trajectory is shown in Figure 2. In both domains, inputs are presented as minibatches of i.i.d. samples with their regression targets. Additional domain details are provided in the appendix.

5.1. Learning sequential plans

In the first experiment we trained a predictron to predict trajectories generated by a simple deterministic policy in 13×13 random mazes with random starting positions. Figure 3 shows the weighted preturns $\mathbf{w}^k \mathbf{g}^k$ and the resulting prediction $\mathbf{g}^{\boldsymbol{\lambda}} = \sum_k \mathbf{w}^k \mathbf{g}^k$ for six example inputs and targets. The predictions are almost perfect—the training error was very close to zero. The full prediction is composed from weighted preturns which decompose the trajectory piece by piece, starting at the start position in the first step $k = 1$, and where often multiple policy steps are added per planning step. The predictron was not informed about the sequential build up of the targets—it never sees a policy

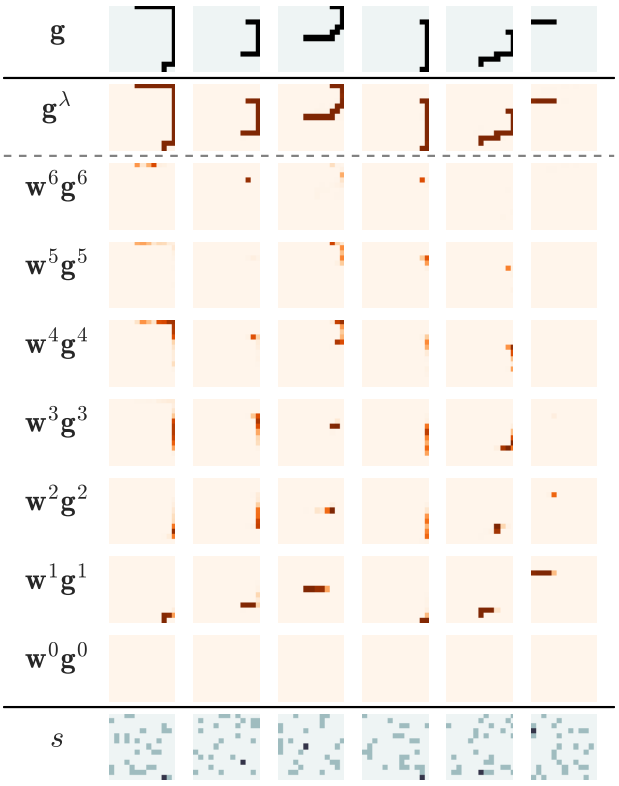

Figure 3. **Indication of planning.** Sampled mazes (grey) and start positions (black) are shown superimposed on each other at the bottom. The corresponding target vector $\mathbf{g}$, arranged as a matrix for visual clarity, is shown at the top. The ensembled prediction $\sum_k \mathbf{w}^k \mathbf{g}^k = \mathbf{g}^{\boldsymbol{\lambda}}$ is shown just below the target—the prediction is near perfect. The weighted preturns $\mathbf{w}^k \mathbf{g}^k$ that make up the prediction are shown below $\mathbf{g}^{\boldsymbol{\lambda}}$. We can see that full predicted trajectory is built up in steps, starting at the start position and then planning through the trajectory in sequence.

walking through the maze, only the resulting trajectories—and yet sequential plans emerged spontaneously. Notice also that the easier trajectory on the right was predicted in only two steps, while more thinking steps are used for more complex trajectories.

5.2. Exploring the predictron architecture

In the next set of experiments, we tackle the problem of predicting connectivity of multiple pairs of locations in a random maze, and the problem of learning many different value functions from our simulator of the game of pool. We use these more challenging domains to examine three binary dimensions that differentiate the predictron from standard deep networks. We compare eight predictron variants corresponding to the corners of the cube on the left in Figure 4.

The first dimension, labelled r, γ, corresponds to whether or not we use the structure of an MRP model. In the MRP case internal rewards and discounts are both learned. In the non-(r, γ) case, which corresponds to a vanilla hidden-to-hidden neural network module, internal rewards and discounts are ignored by fixing their values to $\mathbf{r}^k = \mathbf{0}$ and $\boldsymbol{\gamma}^k = \mathbf{1}$.

The second dimension is whether a K-step accumulator or λ-accumulator is used to aggregate preturns. When a λ-accumulator is used, a λ-preturn is computed as described in Section 3. Otherwise, intermediate preturns are ignored by fixing $\boldsymbol{\lambda}^k = 1$ for $k < K$. In this case, the overall output of the predictron is the maximum-depth preturn $\mathbf{g}^K$.

The third dimension, labelled *usage weighting*, defines the loss that is used to update the parameters $\boldsymbol{\theta}$. We consider two options: the preturn losses can either be weighted uniformly (see Equation 6), or the update for each preturn $\mathbf{g}^k$ can be weighted according to the weight $\boldsymbol{w}^k$ that determines how much it is used in the λ-predictron's overall output. We call the latter loss 'usage weighted'. Note that for architectures without a λ-accumulator, $\boldsymbol{w}^k = 0$ for $k < K$, and $\boldsymbol{w}^K = 1$, thus usage weighting then implies backpropagating only the loss on the final preturn $\mathbf{g}^K$.

All variants utilise a convolutional core with 2 intermediate hidden layers; parameters were updated by supervised learning (see appendix for more details). Root mean squared prediction errors for each architecture, aggregated over all predictions, are shown in Figure 4. The top row corresponds to the random mazes and the bottom row to the pool domain. The main conclusion is that learning an MRP model improved performance greatly. The inclusion of $\boldsymbol{\lambda}$ weights helped as well, especially on pool. Usage weighting further improved performance.

5.3. Comparing to other architecture

Our third set of experiments compares the predictron to feedforward and recurrent deep learning architectures, with and without skip connections. We compare the corners of a new cube, as depicted on the left in Figure 5, based on three different binary dimensions.

The first dimension of this second cube is whether we use a predictron, or a (non-λ, non-(r, γ)) deep network that does not have an internal model and does not output or learn from intermediate predictions. We use the most effective predictron from the previous section, i.e., the (r, γ, λ)-predictron with usage weighting.

The second dimension is whether all cores share weights (as in a recurrent network), or each core uses separate weights (as in a feedforward network). The non-λ, non-(r, γ) variants of the predictron then correspond to standard (convolutional) feedforward and (unrolled) recurrent neural networks respectively.

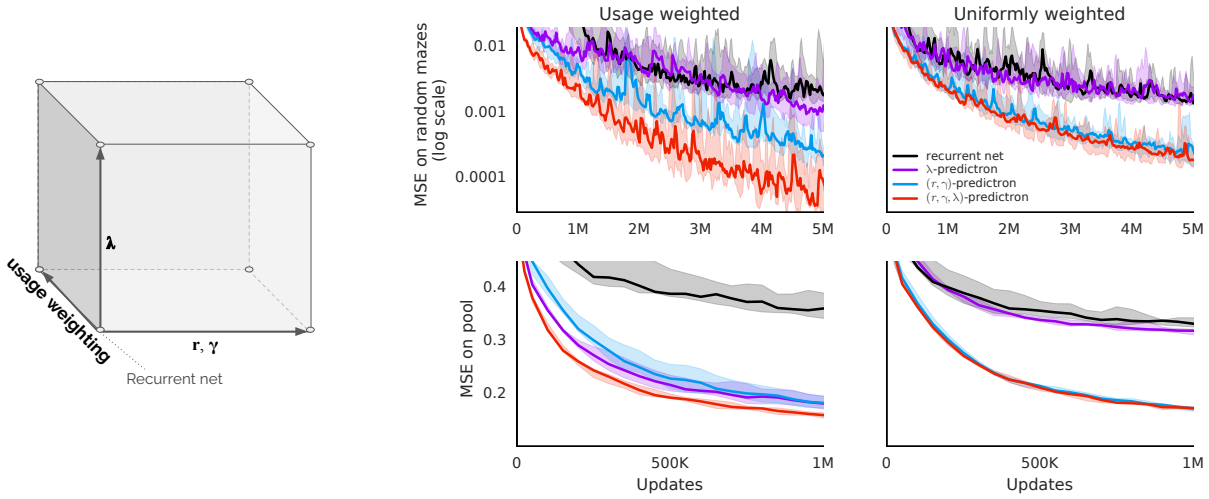

Figure 4. **Exploring predictron variants.** Aggregated prediction errors over all predictions (20 for mazes, 280 for pool) for the eight predictron variants corresponding to the cube on the left (as described in the main text), for both random mazes (top) and pool (bottom). Each line is the median of RMSE over five seeds; shaded regions encompass all seeds. The full (r, γ, λ)-prediction (**red**) consistently performed best.

The third dimension is whether we include skip connections. This is equivalent to defining the model step to output a change to the current state, $\Delta\mathbf{s}$, and then defining $\mathbf{s}^{k+1} = h(\mathbf{s}^k + \Delta\mathbf{s}^k)$, where h is the non-linear function—in our case a ReLU, $h(x) = \max(0, x)$. The deep network with skip connections is a variant of ResNet (He et al., 2015).

Root mean squared prediction errors for each architecture are shown in Figure 5. All (r, γ, λ)-predictrons (red lines) outperformed the corresponding feedforward or recurrent baselines (black lines) both in the random mazes and in pool. We also investigated the effect of changing the depth of the networks (see appendix); the predictron outperformed the corresponding feedforward or recurrent baselines for all depths, with and without skip connections.

5.4. Semi-supervised learning by consistency

We now consider how to use the predictron for semi-supervised learning, training the model on a combination of labelled and unlabelled random mazes. Semi-supervised learning is important because a common bottleneck in applying machine learning in the real world is the difficulty of collecting labelled data, whereas often large quantities of unlabelled data exist.

We trained a full (r, γ, λ)-predictron by alternating standard supervised updates with consistency updates, obtained by stochastically minimizing the consistency loss (8), on additional unlabelled samples drawn from the same distribution. For each supervised update we apply either 0, 1, or 9 consistency updates. Figure 6 shows that the performance improved monotonically with the number of consistency updates, measured as a function of the number of labelled samples consumed.

5.5. Analysis of adaptive depth

In principle, the predictron can adapt its depth to 'think more' about some predictions than others, perhaps depending on the complexity of the underlying target. We saw indications of this in Figure 3. We investigate this further by looking at qualitatively different prediction types in pool: ball collisions, rail collisions, pocketing balls, and entering or staying in quadrants. For each prediction type we consider several different time-spans (determined by the real-world discount factors associated with each pseudo-reward). Figure 7 shows distributions of *depth* for each type of prediction. The 'depth' of a predictron is here defined as the effective number of model steps. If the predictron relies fully on the very first value (i.e., $\boldsymbol{\lambda}^0 = 0$), this counts as 0 steps. If, instead, it learns to place equal weight on all rewards and on the final value, this counts as 16 steps. Concretely, the depth $\boldsymbol{d}$ can be defined recursively as $\boldsymbol{d} = \boldsymbol{d}^0$ where $\boldsymbol{d}^k = \boldsymbol{\lambda}^k(1 + \boldsymbol{\gamma}^k \boldsymbol{d}^{k+1})$ and $\boldsymbol{d}^K = \mathbf{0}$. Note that even for the same input state, each prediction has a separate depth.

The depth distributions exhibit three properties. First, different types of predictions used different depths. Second, depth was correlated with the real-world discount for the first four prediction types. Third, the distributions are not strongly peaked, which implies that the depth can differ per input even for a single real-world discount and prediction type. In a control experiment (not shown) we used a

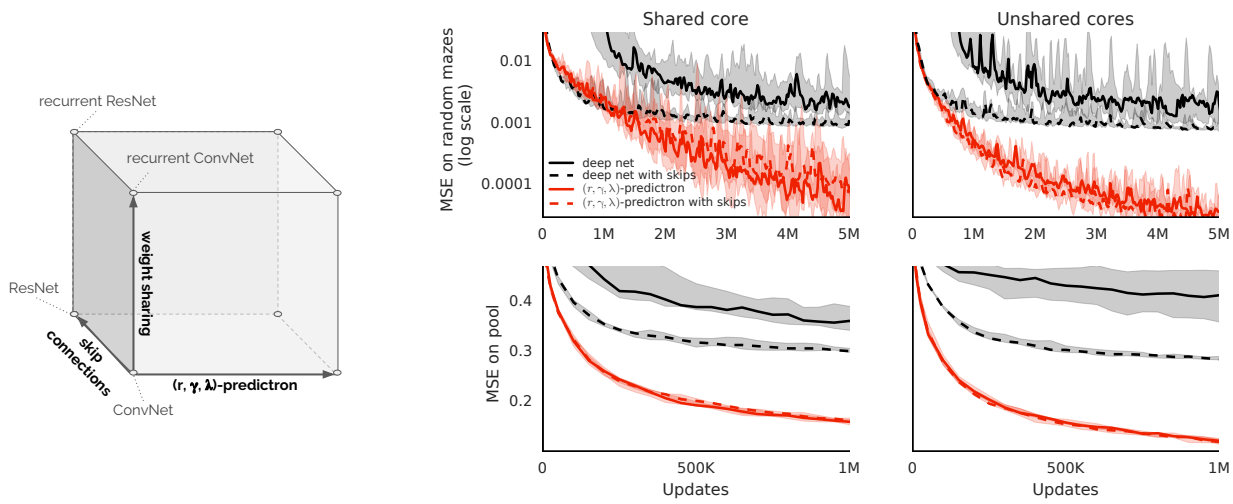

Figure 5. **Comparing predictron to baselines.** Aggregated prediction errors on random mazes (top) and pool (bottom) over all predictions for the eight architectures corresponding to the cube on the left. Each line is the median of RMSE over five seeds; shaded regions encompass all seeds. The full (r, γ, λ)-predictron (**red**), consistently outperformed conventional deep network architectures (**black**), with and without skips and with and without weight sharing.

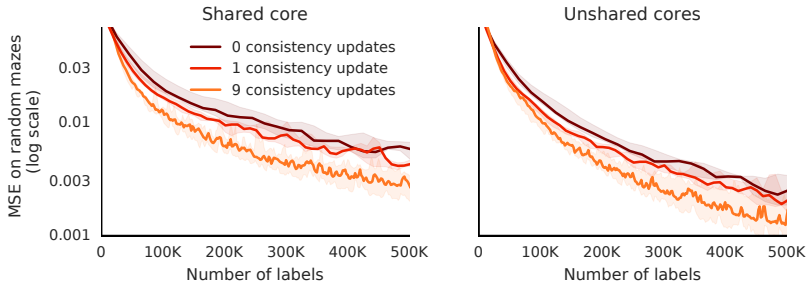

Figure 6. **Semi-supervised learning.** Prediction errors of the (r, γ, λ)-predictrons (shared core, no skips) using 0, 1, or 9 consistency updates for every update with labelled data, plotted as function of the number of labels consumed. Learning performance improves with more consistency updates.

scalar λ shared among all predictions, which reduced performance in all scenarios, indicating that the heterogeneous depth is a valuable form of flexibility.

5.6. Using predictions to make decisions

We test the quality of the predictions in the pool domain to evaluate whether they are well-suited to making decisions. For each sampled pool position, we consider a set I of different initial conditions (different angles and velocity of the white ball), and ask which is more likely to lead to pocketing coloured balls. For each initial condition $s \in I$, we apply the (r, γ, λ)-predictron (shared cores, 16 model steps, no skip connections) to obtain predictions $\mathbf{g}^\lambda$. We ensemble the predictions associated to pocketing any ball (except the white one) with discounts $\gamma = 0.98$ and $\gamma = 1$. We select the condition s^* that maximises this sum.

We then roll forward the pool simulator from s^* and log the number of pocketing events. Figure 2 shows a sampled rollout, using the predictron to pick s^*. When providing the choice of 128 angles and two velocities for initial conditions ($|I| = 256$), this procedure resulted in pocketing 27 coloured balls in 50 episodes. Using the same procedure with an equally deep convolutional network only resulted in 10 pocketing events. These results suggest that the lower loss of the learned (r, γ, λ)-predictron translated into meaningful improvements when informing decisions. A video of the rollouts selected by the predictron is available at the following url: `https://youtu.be/BeaLdaN2C3Q`.

6. Related work

Lee et al. (2015) introduced a neural network architecture where classifications branch off intermediate hidden layers. An important difference with respect to the λ-predictron is that the weights are hand-tuned as hyper-parameters, whereas in the predictron the λ weights are learnt and, more

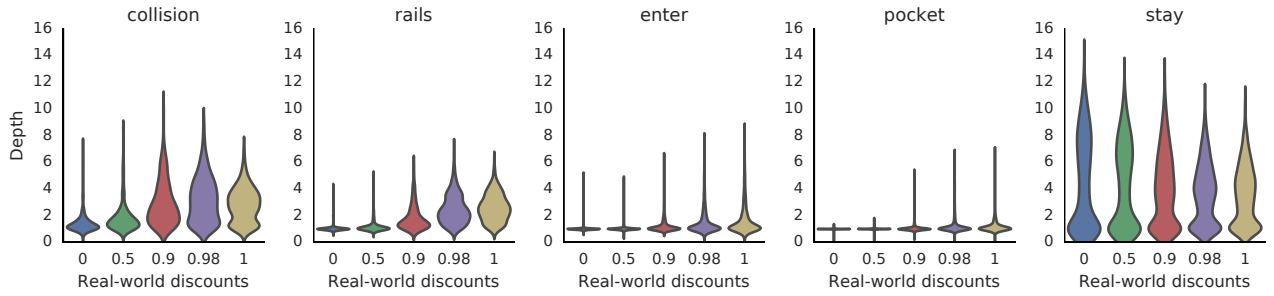

Figure 7. **Thinking depth.** Distributions of thinking depth on pool for different types of predictions and for different real-world discounts.

importantly, conditional on the input. Another difference is that the loss on the auxiliary classifications is used to speed up learning, but the classifications themselves are not combined into an aggregate prediction; the output of the model itself is the deepest prediction.

Graves (2016) introduced an architecture with adaptive computation time (ACT), with a discrete (but differentiable) decision on when to halt, and aggregating the outputs at each pondering step. This is related to our $\boldsymbol{\lambda}$ weights, but obtains depth in a different way; one notable difference is that the λ-predictron can use different pondering depths for each of its predictions.

Value iteration networks (VINs) (Tamar et al., 2016) also learn value functions end-to-end using an internal model, similar to the (non-λ) predictron. However, VINs plan via convolutional operations over the full input state space; whereas the predictron plans via imagined trajectories through an abstract state space. This may allow the predictron architecture to scale much more effectively in domains that do not have a natural two-dimensional encoding of the state space.

The notion of learning about many predictions of the future relates to work on predictive state representations (PSRs; Littman et al., 2001), general value functions (GVFs; Sutton et al., 2011), and nexting (Modayil et al., 2012). Such predictions have been shown to be useful as representations (Schaul & Ring, 2013) and for transfer (Schaul et al., 2015). So far, however, none of these have been considered for learning abstract models.

Schmidhuber (2015) discusses learning abstract models, but maintains separate losses for the model and a controller, and suggests training the model unsupervised to compactly encode the entire history of observations, through predictive coding. The predictron's abstract model is instead trained end-to-end to obtain accurate values.

7. Conclusion

The predictron is a single differentiable architecture that rolls forward an internal model to estimate external values. This internal model may be given both the structure and the semantics of traditional reinforcement learning models. But, unlike most approaches to model-based reinforcement learning, the model is fully abstract: it need not correspond to the real environment in any human understandable fashion, so long as its rolled-forward "plans" accurately predict outcomes in the true environment.

The predictron may be viewed as a novel network architecture that incorporates several separable ideas. First, the predictron outputs a value by accumulating rewards over a series of internal planning steps. Second, each forward pass of the predictron outputs values at multiple planning depths. Third, these values may be combined together, also within a single forward pass, to output an overall ensemble value. Finally, the different values output by the predictron may be encouraged to be self-consistent with each other, to provide an additional signal during learning. Our experiments demonstrate that these differences result in more accurate predictions of value, in reinforcement learning environments, than more conventional network architectures.

We have focused on value prediction tasks in uncontrolled environments. However, these ideas may transfer to the control setting, for example by using the predictron as a Q-network (Mnih et al., 2015). Even more intriguing is the possibility of learning an internal MDP with abstract internal actions, rather than the MRP considered in this paper. We aim to explore these ideas in future work.

References

Bellman, Richard. *Dynamic programming*. Princeton University Press, 1957.

Chiappa, Silvia, Racaniere, Sebastien, Wierstra, Daan, and Mohamed, Shakir. Recurrent environment simulators. 2016.

Glorot, Xavier, Bordes, Antoine, and Bengio, Yoshua. Deep sparse rectifier neural networks. In *Aistats*, volume 15, pp. 275, 2011.

Graves, Alex. Adaptive computation time for recurrent neural networks. *CoRR*, abs/1603.08983, 2016. URL `http://arxiv.org/abs/1603.08983`.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Kingma, Diederik P and Ba, Jimmy. A method for stochastic optimization. In *International Conference on Learning Representation*, 2015.

LeCun, Yann, Bottou, Léon, Bengio, Yoshua, and Haffner, Patrick. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Lee, Chen-Yu, Xie, Saining, Gallagher, Patrick, Zhang, Zhengyou, and Tu, Zhuowen. Deeply-supervised nets. In *AISTATS*, volume 2, pp. 6, 2015.

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. In *ICLR*, 2016.

Littman, Michael L, Sutton, Richard S, and Singh, Satinder P. Predictive representations of state. In *NIPS*, volume 14, pp. 1555–1561, 2001.

Mnih, V, Badia, A Puigdomènech, Mirza, M, Graves, A, Lillicrap, T, Harley, T, Silver, D, and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

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

Modayil, Joseph, White, Adam, and Sutton, Richard S. Multi-timescale nexting in a reinforcement learning robot. In *International Conference on Simulation of Adaptive Behavior*, pp. 299–309. Springer, 2012.

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

Schaul, Tom and Ring, Mark B. Better Generalization with Forecasts. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Beijing, China, 2013.

Schaul, Tom, Horgan, Daniel, Gregor, Karol, and Silver, David. Universal Value Function Approximators. In *International Conference on Machine Learning (ICML)*, 2015.

Schmidhuber, Juergen. 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.

Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

Sutton, R. S. Integrated architectures for learning, planning and reacting based on dynamic programming. In *Machine Learning: Proceedings of the Seventh International Workshop*, 1990.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT press, Cambridge MA, 1998.

Sutton, Richard S. TD models: Modeling the world at a mixture of time scales. In *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 531–539, 1995.

Sutton, Richard S, Modayil, Joseph, Delp, Michael, Degris, Thomas, Pilarski, Patrick M, White, Adam, and Precup, Doina. Horde: A scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pp. 761–768. International Foundation for Autonomous Agents and Multiagent Systems, 2011.

Tamar, Aviv, Wu, Yi, Thomas, Garrett, Levine, Sergey, and Abbeel, Pieter. Value iteration networks. In *Neural Information Processing Systems (NIPS)*, 2016.

Todorov, Emanuel, Erez, Tom, and Tassa, Yuval. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033. IEEE, 2012.

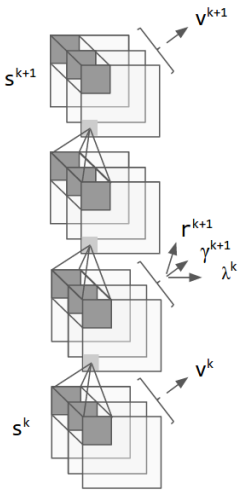

Figure 8. The predictron core used in our experiments.

A. Architecture

The state representation f is a two-layer convolutional neural network (LeCun et al., 1998). There is a *core* c, again based on convolutions, that combines both MRP model and λ-network into a single repeatable module, such that $\mathbf{s}^{k+1}, \mathbf{r}^{k+1}, \boldsymbol{\gamma}^{k+1}, \boldsymbol{\lambda}^k = c(\mathbf{s}^k)$. This core is deterministic, and is duplicated K times in the predictron with shared weights. (The predictron with unshared weights has K distinct cores.) Finally, the value network v is a fully connected neural network that computes $\mathbf{v}^k = v(\mathbf{s}^k)$.

Concretely, the $core$ (Figure 8) consists first of a convolutional layer that maps into an intermediate (hidden) layer. From this layer, another two convolutions compute the next abstract state of the predictron. Additionally, this same hidden layer is flattened and fed into three separate networks, with two fully connected layers each. The outputs of these three networks represent the internal rewards, discounts, and lambdas. A similar small network also hangs off the internal states, in addition to the core, and computes the values. All convolutions use 3×3 filters and a stride of one, and use padding to retain the size of the feature maps. All feature maps have 32 channels. The hidden layers within the MLPs have 32 hidden units.

In Figure 8 the convolutional layers are schematically drawn with three channels, flattening is represented by curly brakets, while the arrows represent the small multi-layer perceptrons which compute values, rewards, discounts and lambdas.

We allow up to 16 model steps in our experiments, resulting in 52-layer deep networks—two convolutional layers for the state representations, $3 \times 16 = 48$ convolutional layers for the core steps, and two fully-connected layers for the values on top of the final state. Between each two layers we apply batch normalization (Ioffe & Szegedy, 2015) followed by a ReLU non-linearity (Glorot et al., 2011). The value and reward networks end with a linear layer, whereas the discount and λ-networks additionally add a sigmoid non-linearity to ensure that these quantities are in $[0, 1]$.

For the illustrative maze experiment in Section 5.1, a smaller network architecture is employed with 6 model steps and convolutional feature maps of 16 channels. Additionally, the subnetworks to compute values, rewards, discounts, and lambdas are composed of a 1×1 convolution with a stride of 1 and 8 channels before a fully connected hidden layer of size 128. The rest of network architecture is as described above.

B. Training

All experiments used the supervised (Monte-Carlo) update described in Section 4 except for the semi-supervised experiment which used the consistency update described in Section 4.1. We update all parameters by applying the Adam optimiser (Kingma & Ba, 2015) to stochastic gradients of the corresponding loss functions. Each return is normalised by dividing it by its standard deviation (as measured, prior to the experiment, on a set of 20,000 episodes). In all experiments, the learning rate was 0.001, and the other parameters of the Adam optimiser were $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. We used mini-batches of 100 samples.

C. Comparing architectures of different depths

We investigated the effect of changing the depth of the networks, with and without skip connections. Figure 9 in shows that skip connections (dashed lines) make the conventional architectures (black/grey lines) more robust to the depth (i.e., the black/grey dashed lines almost overlap, especially on pool), and that the predictron outperforms the corresponding feedforward or recurrent baselines for all depths, with and without skips.

D. Capacity comparisons

In this section, we present some additional experiments comparing the predictron to more conventional deep networks. The purposes of these experiments are 1) to show that the conclusions obtained above do not depend on the precise architecture used, and 2) to show that the structure of the network—whether we use a predictron or not—is more important than the raw number of parameters.

Specifically, we again consider the same 20 by 20 random mazes, and the pool task described in the main text. As described in Section A, for the results in the paper we used an encoder that preserved the size of the input plans, 20×20 for the mazes and 28×28 for pool. Each convolution had 32 channels and therefore the abstract states were $20\times20\times32$ for the mazes and $28\times28\times32$ for pool.

We now consider a different architecture, where we no longer pad the convolutions used in the encoder. For the mazes, we still use two layers of 3×3 stride-1 convolutions, which means the planes reduce in size to 16×16. This means that the abstract states are about one third smaller. For pool, we use three 5×5 stride-1 convolutions, which bring us from 28×28 down to 16×16 as well. So, the abstract states are now of equal size for both experiments. For pool, this is approximately a two-thirds reduction, which helps reduce the compute needed to run the model.

Most of the parameters in the predictron are in the fully connected layers. Previously, the first fully connected layer for each of the internal values, rewards, discounts, and λ-parameters would take a flattened abstract state, and then go into 32 hidden nodes. This means the number of parameters in this layer were $20 \times 20 \times 32 \times 32 = 409,600$ for the mazes and $28 \times 28 \times 32 \times 32 = 802,816$

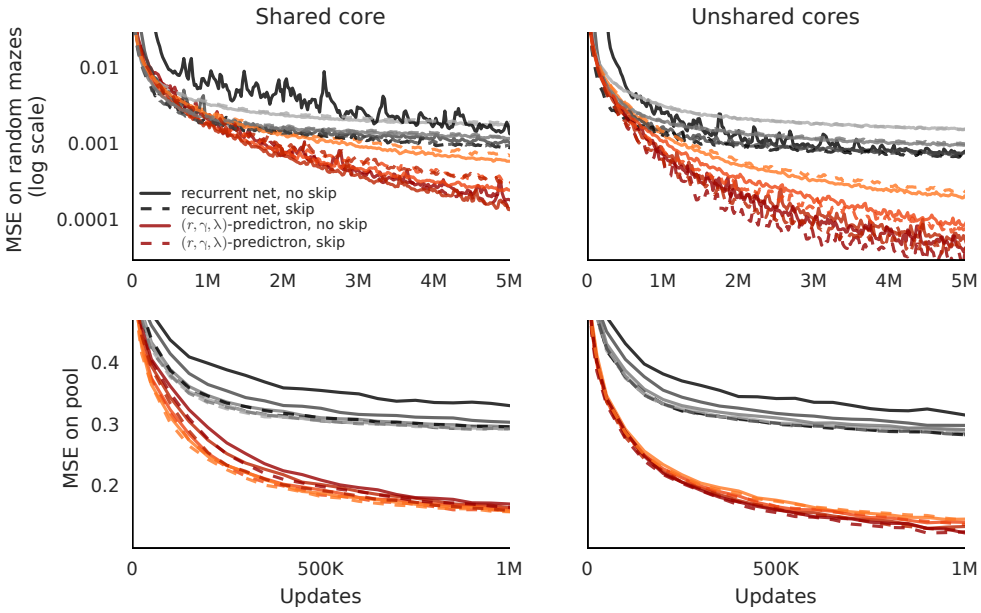

Figure 9. **Comparing depths.** Comparing the (r, γ, λ)-predictron (**red**) against more conventional deep networks (**black**) for various depths (2, 4, 8, or 16 model steps, corresponding to 10, 16, 28, or 52 total layers of depth). Lighter colours correspond to shallower networks. Dashed lines correspond to networks with skip connections.

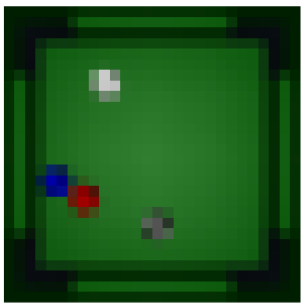

Figure 10. **Pool input frame.** An example of a 28x28 RGB input frame in the pool domain.

for pool. The predictron with shared core would have four of these layers, one for each of the internal values, rewards, discounts, and λs, compared to one for the deep network which only has values. We change this in two ways. First, we add a 1×1 convolution with a stride of 1 and 8 channels before the first fully connected layer for each of these outputs. This reduces the number of channels, and therefore the number of parameters in the subsequent fully-connected layer, by one fourth. Second, we tested three different numbers of hidden nodes: 32, 128, or 512.

The deep network with 128 hidden nodes for its values has the exact same number of parameters as the (r, γ, λ)-predictron with 32 hidden nodes for each of its outputs. Before, the deep network had fewer parameters, because we kept this number fixed at 32 across experiments. This opens the question of whether the improved performance of the predictron was not just an artifact of having more parameters. We tested this hypothesis, and the results are shown in Figure 11.

Figure 11 shows that in each setting—on the mazes and pool, and with or without shared cores—both. The predictrons always performed better than all the deep networks. This includes the 32 node predictron (darkest red) compared to the 512 node deep network (lightest blue), even though the latter has approximately 4 times as many parameters (1.27M vs 4.85M). This means that the number of parameters mattered less than whether or not we use a predictron.

E. Additional domain details

We now provide some additional details of domains.

E.1. Pool

To generate sequences in the Pool domain, the initial locations of 4 balls of different colours are sampled at random. The white ball is the only one moving initially. Its velocity has a norm sampled uniformly between 7 and 14. The initial angle is sampled uniformly in the range $(0, 2\pi)$. From the initial condition, the Mujoco simulation is run forward until all balls have stopped moving; sequences that last more than 151 frames are rejected, and a new one is generated as replacement. Each frame is rendered by Mujoco as a 280x280 RGB image, and subsequently downsampled through bilinear interpolation to a 28x28 RGB input (see Figure 10 for an example). Since the 280 signals described in Section 6.1 as targets for the Pool experiments have very different levels of sparsity, resulting in values with very different scales, we have normalised the pseudo returns. The normalization procedure consisted in dividing all targets by their standard deviation, as empirically measured across an initial set of 20,000 sequences.

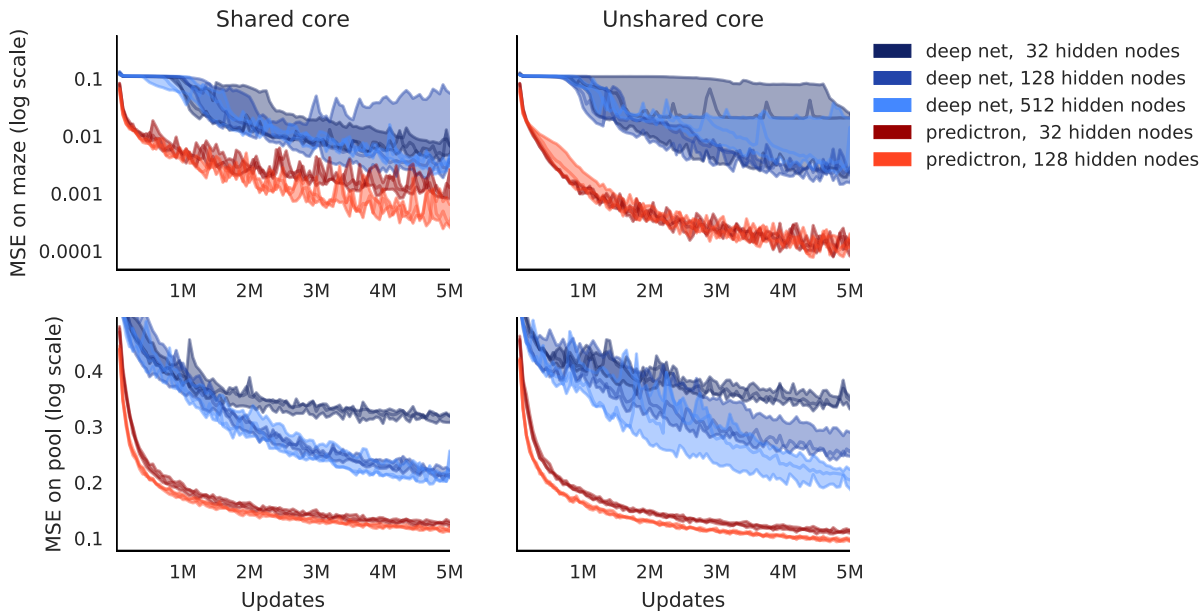

Figure 11. **Comparing depths.** Comparing the (r, γ, λ)-predictron (**red**) against more conventional deep networks (**blue**) for different numbers hidden nodes in the fully connected layers, and therefore different total numbers of parameters. The deep networks with 32, 128, and 512 nodes respectively have 381,416, 1,275,752, and 4,853,096 parameters in total. The predictrons with 32 and 128 nodes respectively have 1,275,752, and 4,853,096 parameters in total. Note that the number of parameters for the 32 and 128 node predictrons are exactly equal to the number of parameters for the 128 and 512 node deep networks.

E.2. Random Mazes

E.2.1. First Task

The mazes are generated by ensuring that around 15% of locations are walls. The policy takes as observation the wall configuration in four locations adjacent to its position and maps each of these configurations to an action. For each maze, the policy is stepped for 60 steps from a uniformly random start location. The target $\mathbf{g}$ indicates whether the trajectory has traversed each maze location.

E.2.2. Second Task

To generate mazes we first determine, with a stochastic line search, a number of walls so that the top-left corner is connected to the bottom-right corner (both always forced to be empty) in approximately 50% of the mazes. We then shuffle the walls uniformly randomly. For 20 by 20 mazes this means 70% of locations are empty and 30% contain walls. More than a googol different such 20-by-20 mazes exist (as $\binom{398}{120} > 10^{100}$).