Successor representation (SR) and replay

cyril regan

May, 2021

1 Introduction

This objective of this report is to sum up the mathematical assumptions of the Successor Representation (Dayan, 1993; Gershman, 2018) with the formalism of Reinforcement Learning, as introduced in (Sutton-Barto, 1998).

1.1 MDP

A reinforcement learning task that satisfies the Markov property is called a Markov Decision Process, or MDP. A particular finite MDP is defined by its state and action sets and by the one-step dynamics of the environment. Hence, the probability of transitioning to $s_{t+1} = s$, given the previous 'history', is fully captured by conditioning only on the current state $s_t = s$ and action $a_t = a$.

In other words, the future is independent of the past, given the present. This is described by the transition function:

$$T(s'|s,a) = \mathcal{P}^a_{ss'} = \mathbb{P}[s_{t+1} = s'|s_t = s, a_t = a] \tag{1}$$

Similarly, given any current state and action, s and a, together with any next state s', the expected value of the next reward is :

$$\mathcal{R}^a_{ss'} = \mathbb{E}[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'] \tag{2}$$

This characterization of the reward dynamics of an MDP in terms of $\mathcal{R}^a_{ss'}$ is slightly unusual. It is more common in the MDP literature to describe the reward dynamics in terms of the expected next reward given just the current state and action, i.e., by

$$r_{t+1} = \mathcal{R}^a_s = r(s,a) = \mathbb{E}[r_{t+1}|s_t = s, a_t = a] = \sum_{s'} \mathcal{P}^a_{ss'}\mathcal{R}^a_{ss'} \tag{3}$$

The 4-tuple $(e_t = s_t, a_t, r_t, s_{t+1})$ is called a target memory, a transition, or an experience (e_t).

1.1.1 Policy

At any given state, the agent undergoes a decision-making process in order to select an action. The policy is a function that maps the state of an agent to an action. This can either be deterministic :

$$a = \pi(s) \tag{4}$$

or stochastic :

$$\pi(a|s) = \mathbb{P}[a_t = a|s_t = s] \tag{5}$$

1.1.2 Value functions

Value functions on policy : Reinforcement learning is concerned with the estimation of the *state-value function* $V(s)$ or *action-value function* $Q(s, a)$, the total reward an agent expects to earn in the future, with short-term rewards weighed more highly than long-term rewards. Of course the rewards the agent can expect to receive in the future depends on what actions it will take. Accordingly, value functions are defined with respect to particular policies. The state-value function is defined as the expected discounted future return following the policy π (Sutton-Barto, 1998):

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...|s_t = s] \\ &= \mathbb{E}_\pi[\sum_{k=0}^{+\infty} \gamma^k r_{t+k}|s_t = s] \end{aligned} \tag{6}$$

where γ is a discount factor that captures a preference for proximal rewards. Similarly the action-value function is defined by :

$$Q^\pi(s, a) = \mathbb{E}_\pi[\sum_{k=0}^{+\infty} \gamma^k r_{t+k}|s_t = s, a_t = a] \tag{7}$$

State-value function can be written recursively with Bellman equation :

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{+\infty} \gamma^k r_{t+k}|s_t = s\right] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma \sum_{k=0}^{+\infty} \gamma^k r_{t+k+1}|s_t = s] \\ &= \sum_a \pi(s, a)\left[r_{t+1} + \sum_{s'} \mathcal{P}^a_{ss'} \gamma\, \mathbb{E}_\pi\left[\sum_{k=0}^{+\infty} \gamma^k r_{t+k+1}|s_{t+1} = s'\right]\right] \\ &= \sum_a \pi(s, a)\left[r_{t+1} + \gamma \sum_{s'} \mathcal{P}^a_{ss'} V^\pi(s')\right] \\ &= \sum_a \pi(s, a)\left[r(a, s) + \gamma \sum_{s'} \mathcal{P}^a_{ss'} V^\pi(s')\right] \end{aligned} \tag{8}$$

Value functions on optimal policy : A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $V^{\pi}(s) \geq V^{\pi'}(s)$ for all state $s \in \mathcal{S}$

All the optimal policies by denoted by π^* share the same state-value function V^*.

$$V^*(s) = \max_{\pi} V^{\pi}(s) \qquad \forall s \in \mathcal{S} \tag{9}$$

The optimal policies share also the same action-value function Q^* :

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s) \qquad \forall s \in \mathcal{S} \text{ and } a \in \mathcal{A}(s) \tag{10}$$

Thus, we can write Q^* in terms of V^* as follows:

$$Q^*(s,a) = \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a] \tag{11}$$

Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} V^*(s) &= \max_a Q^*(s,a) \\ &= \max_a \left[r(a,s) + \gamma \sum_{s'} \mathcal{P}^a_{ss'} V^*(s') \right] \end{aligned} \tag{12}$$

$$Q^*(a,s) = r(a,s) + \gamma \sum_{s'} \mathcal{P}^a_{ss'} \max_{a'} Q^*(s',a') \tag{13}$$

1.2 Model-based algorithms

Model-based decision algorithms are explicitly based on the underlying "model" (i.e., the reward function r and the transition function $\mathcal{P}^a_{ss'}$), which is either given a priori or learned. As the model is known, an estimate of the value functions can be updated by :

$$\hat{V} \leftarrow r(s) + \gamma \sum_{s'} \max_a \mathcal{P}^a_{ss'} \hat{V}(s') \tag{14}$$

$$\hat{Q}(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} \mathcal{P}^a_{ss'} \max_{a'} \hat{Q}(s',a'), \tag{15}$$

Where $r(s) = \max_a r(s,a)$. Figure 1 illustrates the model-based schema.

Drawback: this architecture is computationally expensive, because value estimation are computed by planning (i.e. propagating the known rewards in the graph generated by the transition function) and are used to decide an action in each state.

Advantages : Compared to Model-free algorithms (described below) Model-Based algorithm uses a reduced number of interactions with the real environment during the learning phase. Moreover, the agent endowed with a model requires only a small amount of experience to adapt to changes in rewards localisation or in environment structure.

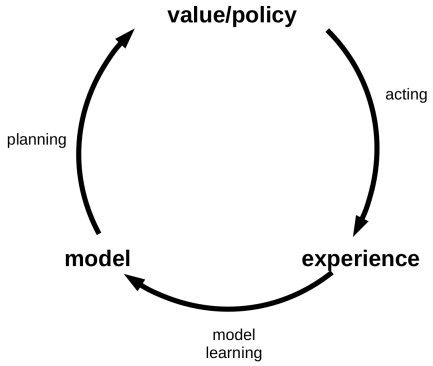


Figure 1: Model-based schema

1.3 Model-free algorithms

Model-free algorithms directly estimate the value function V from experienced transitions and rewards, without explicitly using or learning a model. In a deterministic setting, the estimated value functions $\hat{V}$ or $\hat{Q}$ can be represented by a lookup table storing the estimates for each state (resp state-action) while physically interacting with the environment, through temporal difference (TD) learning.

$$\hat{V}(s) \leftarrow r(s) + \gamma \hat{V}(s'), \tag{16}$$

$$\hat{Q}(s,a) \leftarrow r(s,a) + \gamma \max_{a'} \hat{Q}(s',a'), \tag{17}$$

Where s' corresponds to the successor state following the policy π of the Model-free algorithms.

The TD error δ is :

$$\delta = r(s) + \gamma \hat{V}(s') - \hat{V}(s) \tag{18}$$

$$\delta = r(s,a) + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a) \tag{19}$$

And the update of the value function is done by :

$$\hat{V}^{new}(s) = \hat{V}^{old}(s) + \alpha.\delta, \tag{20}$$

$$\hat{Q}^{new}(s,a) = \hat{Q}^{old}(s,a) + \alpha.\delta, \tag{21}$$

where α is the learning rate. Figure 2 illustrates the model-free schema.

Drawback: inflexibility because a local change in the transition or reward functions will produce non-local changes in the value function. Model-free are computationally expensive to train.

Advantages: Trained, a model-free model is computationally efficient to maximize the cumulative reward.

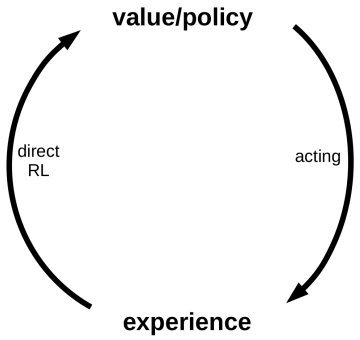


Figure 2: Model-free schema

1.4 Dyna

In order to accommodate for the aforementioned duality of both model-free and model-based strategies in the brain, the Dyna architecture (Sutton, 1990) has been used as a RL paradigm to encapsulate both learning systems. It builds upon traditional model-free techniques where the agent learns directly from real online interactions with the environment. In addition to this, the Dyna framework involves the agent learning a model of the world and allows for experiences to be simulated (similar to hippocampal replay) in order to perform model-based 'planning' steps while the agent is offline.

However, the traditional Dyna architecture proposes an algorithm involved n offline planning steps after each transition and selecting previously experienced target state-action pairs uniformly at random. This prioritization technique which does not place emphasis is suboptimal, because lots of these replayed experiences contain no value information.

The model based planning can be a construction of the successor representation instead to learn the transition state function. The issue is the *prioritization* of experiences to replay while avoiding temporal correlated experiences (correlated data) for the learning of the agent. Figure 3 illustrates the model-based schema.

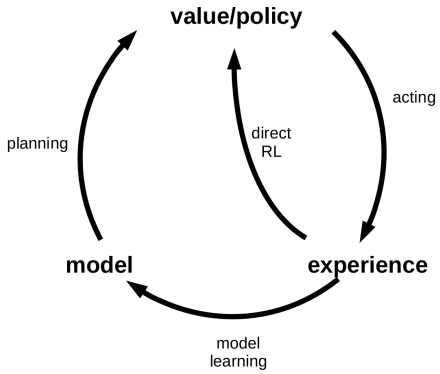


Figure 3: Dyna Model schema

2 The successor representation

The successor representation $\boldsymbol{M}$ represent the "state occupancy". In fact, one can also see the successor representation as a "timeless" state representation. Indeed, the $\boldsymbol{M}$ matrix is a kind of map representing the occupancy of the actual state s and the projection of all future state occupancy (discounting by the γ factor) which could depend also of the action a taken at the actual state s. The construction of the successor representation $\boldsymbol{M}$ with the transition state matrix $\boldsymbol{\mathcal{P}}^a_{ss'}$ is meaningful :

$$\boldsymbol{M}(s', s, a) = \mathbf{1}_{s,s'} + \gamma \mathcal{P}^a_{ss'} + \gamma^2 (\mathcal{P}^2)^a_{ss'} + \gamma^3 (\mathcal{P}^3)^a_{ss'} + ... \tag{22}$$

Here, $\boldsymbol{M}(s', s, a)$ represents the occupancy of state s' knowing the agent is presently in the state s and takes the action a. Let's take a look of the first term of the equation :

$$\begin{aligned} \mathbf{1}_{s,s'} &= 1 \text{ if } s' = s \\ \mathbf{1}_{s,s'} &= 0 \text{ if } s' \neq s \end{aligned} \tag{23}$$

$\mathbf{1}_{s,s'}$ represents trivially the state occupancy of the "present time" which is null for all the states different from the actual.

Then the state occupancy of the first upcoming time is the probability to be in the state s' knowing the agent is in the state s and takes the action a. It is by definition the probability expressed by the state transition matrix discounting by the γ factor :

$$\gamma \boldsymbol{\mathcal{P}}^a_{ss'} \tag{24}$$

But we can go on and project the state occupancy of the next upcoming time :

$$\gamma^2 (\boldsymbol{\mathcal{P}}^2)^a_{ss'} \tag{25}$$

To sum up, the successor representation is a cumulative probability to be in a state from the actual and all future projected times. To say it differently,

$\boldsymbol{M}$ is defined as the discounted occupancy of state s, averaged over all possible trajectories initiated in state s. The SR can intuitively be thought of as a predictive map that encodes each state in terms of the other states that will be visited in the future.

Of course, the future state occupancy is only a "projection" of the present knowledge of the world expressed by the transition matrix $\boldsymbol{\mathcal{P}}^a_{ss'}$. As the manipulation of a sum is not easy in a computational exploitation, one can express the sum with a straightforward limited development :

$$\boldsymbol{M}(s', s, a) = \frac{1}{1 - \gamma \boldsymbol{\mathcal{P}}^a_{ss'}} \tag{26}$$

or in a matrix way :

$$\boldsymbol{M}(a) = (1 - \gamma \boldsymbol{\mathcal{P}}^a)^{-1} \tag{27}$$

The successor representation is also popular because of its nice property which represents any value function as :

$$V(s) = \sum_{s'} \boldsymbol{M}(s, s') \boldsymbol{r}(s') \tag{28}$$

$$Q(s, a) = \sum_{s'} \boldsymbol{M}(s, s', a) \boldsymbol{r}(s', a) \tag{29}$$

Moreover, it is not necessarily to calculate directly the transition matrix $\boldsymbol{\mathcal{P}}^a_{ss'}$ to get the successor representation $\boldsymbol{M}$. An evaluation $\hat{\boldsymbol{M}}$ can be learned via TD-learning for example with the δ_t error :

$$\delta_t(s') = \mathbb{I}[s_t = s'] + \gamma \hat{\boldsymbol{M}}(s_{t+1}, s') - \hat{\boldsymbol{M}}(s_t, s') \tag{30}$$

Notice that unlike the temporal difference error for value learning, the temporal difference error for SR learning is vector-valued, with one error for each successor state.

Advantage on flexibility : changes in the reward function will immediately propagate to all the action-value function. *Drawback* : this is not true of changes in the transition function

3 Replay

A simple way to improve algorithms is to introduce experience replay (Lin, 1992). It consists of storing experiences $e_k = (s_k, a_k, r_k, s'_k)$ while interacting with the environment, and replaying them off-line to accelerate learning. This was demonstrated in the Deep Q-Network (DQN) algorithm (Mnih, 2013), (Mnih, 2015), and was improved by the use of prioritized experience replay (Schaul, 2016).

Here, we introduce two another type of replay which are **simulated** by a model based algorithm (like hippocampal replay). They are model-based experience replay.

The first model presented is based on the *Expected Value of a Bellman backup* (EVB) (Mattar-Daw, 2018), which considers which experience a rational agent ought to replay. The prioritized rule is computed as the product of a *gain* and a *need* term on all possible experiences. The former considers how much the agent could gain by replaying each event, whereas the latter measures how much it need be replayed given its immediate relevance.

The second is a model-based bidirectional search (Khamassi-Girard, 2020) where *prioritized sweeping* and *trajectory sampling* are used sequentially. Prioritized sweeping ranks experiences by how "surprising" they were. Trajectory sampling favors the update of states that have high probability of being visited. It is an heuristic method for prioritized replay, but has the advantage to not compute all possible experiences for each planning step.

3.1 The Expected Value of a Bellman backup (EVB)

The algorithm in (Mattar-Daw, 2018) is based on Dyna architecture (section 1.4) and prioritized Bellman backup. A Bellman backup is the Temporal Difference learning of the action-value Q as eq. (21) :

$$Q(s_k, a_k) \leftarrow Q(s_k, a_k) + \alpha \left[r_k + \gamma \max_a Q(s'_k, a) - Q(s_k, a_k) \right] \tag{31}$$

Bellman backups are performed automatically after each transition in real experience and may also be performed nonlocally during simulated experience. More specifically, the authors suggest that the order of memories to be replayed (in offline learning) should be based upon their "expected value of a Bellman backup" (EVB) :

$$EVB(s_k, a_k) = Gain(s_k, a_k) \times Need(s_k) \tag{32}$$

EVB is the improvement in expected return due to a policy change, and can be calculate by the product of the gain accrued each time the agent visits state s_k (Gain term) and the expected number of times s_k will be visited (Need term). However, EVB is computed on all possible experiences for each planning offline step which is biologically unfeasible, see algorithm 2. The offline learning is imposed arbitrary on the start and the end of each episode with a limited fixed number of planning steps.

3.1.1 Gain term

The gain term quantifies the improvement in expected discounted future rewards as a result of a policy change at the target state-action pair. It is defined as:

$$Gain(s_k, a_k) = \sum_a Q_{\pi_{new}}(s_k, a) \pi_{new}(a|s_k) - \sum_a Q_{\pi_{new}}(s_k, a) \pi_{old}(a|s_k) \tag{33}$$

where $\pi_{new}(a|s_k)$ represents the probability of selecting action a in state s_k after the Bellman backup, and $\pi_{old}(a|s_k)$ is the same quantity before the Bellman

backup. From the state s, the agent selects the next action from a with a fixed policy : the probability distribution over actions in state s computed from the Q-values with a softmax function:

$$\pi(a|s) = \frac{e^{\beta Q(s,a)}}{\sum_i e^{\beta Q(s,i)}} \tag{34}$$

where β is called the inverse temperature which regulates the exploration/exploitation trade-off.

$Gain(s_k, a_k)$ is resistant to updates in Q values unless they also result in a policy change. The term is computed in a manner that is reminiscent of prioritized sweeping, although in this case, positive and negative reward prediction errors have asymmetric roles depending on whether there is more or less reward availability in the rest of the environment.

3.1.2 Need term

The need term quantifies the discounted number of times that an agent expects to visit states in the future, given a target state:

$$Need(s_k) = \sum_i^{\infty} \gamma^i . 1_{S_{t+i,s_k}} \tag{35}$$

It is estimated through the use of the successor representation, which is simply the discounted geometric sum of the transition matrix calculated by eq. (27). Ultimately, the need term prioritizes adjacent states compared to those further into the future, as a result of temporal discounting. The need term is thus obtained directly from the n^{th} row of the SR matrix. Additionally, after learning a policy, the need term maps out probable trajectories that the agent will take, as it focuses on states that are likely to be visited in the future.

3.1.3 Identification

Authors classify each individual backup as forward or reverse by examining the next backup in the sequence. When a backed-up action is followed by a backup in that action's resulting state, it is classified as forward. In contrast, when the state of a backup corresponds to the outcome of the following backed-up action, it is classified as reverse.

3.1.4 EVB Algorithm

Algorithms 1 and 2 are presented below :

Algorithm 1 EVB

procedure EVB

— *Initialisation* —

$planExp \leftarrow [,]$

for each $(s, a) \in \mathcal{S} \times \mathcal{A}$ **do** ▷ *Loop over all state-action of the environment*

Get r (reward),s' (next state) from the environment

Put (s, a, r, s') in $planExp$

▷ *planExp is filled by all the 4-tuples experiences* (s_k, a_k, r_k, s'_k) *with* $k \in range(|\mathcal{S}| \times |\mathcal{A}|)$

Construct the transition matrix $\boldsymbol{\mathcal{P}}$

$M = (1 - \gamma \boldsymbol{\mathcal{P}})^{-1}$ ▷ *Compute the Successor Representation Matrix*

$n_{ep} = 100$ ▷ *number of episodes*

$p_{plan} = 20$ ▷ *number of planning offline steps*

while $n <= n_{ep}$ **do**

— *Planning at the* ***start*** *of episode* —

$Q \leftarrow EVB_{planning}(Q, planExp)$

— *Online Qlearning* —

repeat

$a_t \leftarrow argmax_a softmax_\beta(Q(s_t, a))$ ▷ *policy softmax (Q)*

Take action a_t receive $(s_{t+1}, r_{t+1}, done)$

$\boldsymbol{\mathcal{P}}[s_t, s_{t+1}] \leftarrow \boldsymbol{\mathcal{P}}[s_t, s_{t+1}] + \alpha(1 - \boldsymbol{\mathcal{P}}[s_t, s_{t+1}])$ ▷ *Update transition matrix*

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a'} Q(s'_t, a') - Q(s_t, a_t) \right]$ ▷ *Online Belman Backup on Qvalue*

until $done == True$

— *Planning at the* ***end*** *of episode* —

$Q \leftarrow EVB_{planning}(Q, planExp)$

$n \leftarrow n + 1$

Algorithm 2 EVB planning

procedure $EVB_{planning}$
  while $p <= p_{plan}$ **do**
    for each $(s_k, a_k, r_k, s'_k) \in planExp$ **do**
      for $a \in \mathcal{A}$ **do**
        $Q_{old}(s_k, a) = Q(s_k, a)$
        $\pi_{old}(a|s_k) = softmax(Q_{old}(s_k, a))$ ▷ *Get action probability from* $Q_{old}(s_k, a)$
        $Q_{new}(s_k, a) \leftarrow Q_{old}(s_k, a) + \alpha\left[r_k + \gamma \max_{a'} Q_{old}(s'_k, a') - Q_{old}(s_k, a)\right]$ ▷ *Calcul* Q_{new} *with Bellman backup on* $Q_{old}(s_k, a)$
        $\pi_{new}(a|s_k) = softmax(Q_{new}(s_k, a))$ ▷ *Get action probability from* $Q_{new}(s_k, a)$
        $Gain(s_k, a) = Q_{new}(s_k, a).\pi_{new}(a|s_k) - Q_{new}(s_k, a).\pi_{old}(a|s_k)$ ▷ *Compute the Gain term*
      $M = (1 - \gamma \boldsymbol{\mathcal{P}})^{-1}$ ▷ *Compute (update) the SR matrix*
      $Need(s_k) = M(s_k)$ ▷ *Compute the Need term*
      $EVB(s_k, a_k) = Gain(s_k, a).Need(s_k)$ ▷ *Compute EVB term*
    $k_m \leftarrow argmax_k EVB(s_k, a_k)$
    $(s, a, r, s') \leftarrow planExp(k_m)$ ▷ *Get the experience with the max EVB*
    $Q(s, a) \leftarrow Q(s, a) + \alpha\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$ ▷ *Belman backup on prioritized replay*
    $p \leftarrow p + 1$

3.2 Bidirectionnal model-based approach

In (Cazé-Khamassi, 2018) and (Khamassi-Girard, 2020), the authors postulated a framework for prioritized replay with a heuristic-based approach. Prioritized sweeping (PS) proposes to visit the states starting from those whose Q-value has changed recently, and then to their predecessors, the predecessors of their predecessors, and so on. Trajectory sampling proposes to generate continuous trajectories from the current position until reward is reached, with the idea that it will favor the update of relevant states (avoiding wasting resources on states that have a very low probability of being visited).

3.2.1 Prioritized Sweeping (PS)

Prioritized sweeping (PS) consists of weighing experiences based on how surprising they are. PS also has a tendency to mediate *reverse* or *backward* replay. Specifically, this is measured by the absolute value of the TD error signal

$$\delta = |Q^{n+1}(s,a) - Q^n(s,a)|, \tag{36}$$

with :

$$Q(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} \mathcal{P}^a_{ss'} \max_{a'} Q(s',a'), \tag{37}$$

described in eq. (15).

if δ is higher than a certain threshold, *all possible predecessors* of the current state are added to a priority queue $PQueue$ (that will be used during the PS), with a priority equal to δ, *attenuated by* γ and by *the probability to effectively reach state* s from the predecessor under consideration.

3.2.2 Trajectory Sampling (TS)

The second component is trajectory sampling (TS) from the current location (i.e., the current online estimated position of the agent), which involves accessing memories sequentially along a trajectory of successor states, and drives *forward* replay. These method in (Cazé-Khamassi, 2018) allowed the agent to autonomously decide when, where and for how long it should stop to perform replay within the environment.

3.2.3 Bidirectionnal model-based Algorithm

The online part of the algorithm 3 is quite classical. Given a state s, the softmax function policy choose the action a. Then, given the observed reward r and the new state s', the transition matrix is updated and then the $Q - valueQ(s,a)$ in a model-based manner.

The world model in (Khamassi-Girard, 2020) uses a very basic statistical approach for the transition function T which consists in simply counting how many times (s,a) was followed by s' normalized by the total number of times (s, a) was encountered. Besides, updating the reward function R in the world

model is here based on the latest feedback only. After this step, we measure how much the absolute value of the Q-value has been changed according to:

$$\delta = |Q_{t+1}(s,a) - Q_t(s,a)| \tag{38}$$

if δ is higher than a certain threshold, all possible predecessors of the current state are added to a priority queue $PQueue = \delta$ attenuated by γ and the probability to effectively reach state s from the predecessor under consideration.

After *each* action performed by the agent in the environment, an offline reactivation phase starts. It consists on a series of PS followed by series of TS. The model-based mental traveling is stopped when forward and backward search process have reached a connection state. The schematic representation of the operation of the bidirectional inference algorithm is presented in Figure 4 from (Khamassi-Girard, 2020).

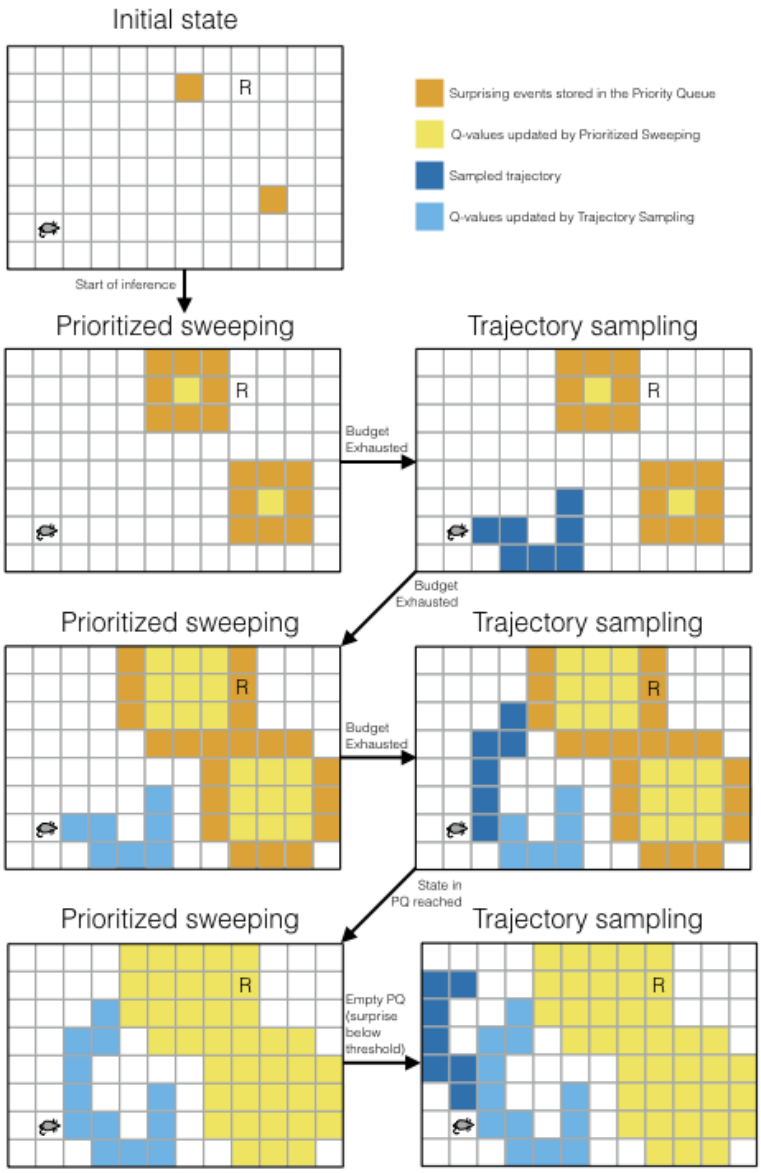


Figure 4: Prioritized sweeping and trajectory sampling phases alternate until an inference stop criterion is reached (Q-value convergence, exhaustion of a general budget, etc.). The mouse represents the agent's position, and R the reward location. Each PS phase stops either when budget is exhausted or when no element in the priority queue has a priority above a fixed threshold. Each TS phase stops either when budget is exhausted or when the trajectory reaches a state stored in the priority queue. Picture from (Khamassi-Girard, 2020) available at https://doi.org/10.6084/m9.figshare.8306132

Algorithm 3 bidirectional algorithm

procedure ONLINE(s_0, Q) ▷ *initial state, Q-values*
  $nbActions \leftarrow 0$
  $s_t \leftarrow s_0$
  $PQueue \leftarrow \{\}$ ▷ *PQueue: empty priority queue*
  $T \leftarrow 0$ ▷ *T: crude transition statistics storage*
  $R \leftarrow 0$ ▷ *R: reward function*
  repeat
    $a_t \leftarrow draw(softmax_\beta(Q(s_t))$ ▷ *policy softmax depending on Qvalue*
    $nbActions \leftarrow nbActions + 1$
    Take action a_t receive s_{t+1}, r_{t+1}
    $T(s_t, a_t, s_{t+1}) \leftarrow T(s_t, a_t, s_{t+1}) + 1$▷ *pseudo transition matrix update in counting the (s_t, a_t, s_{t+1}) occurancy as deterministic way*
    $R(s_t, a_t) \leftarrow r_{t+1}$
    $Q_{old} \leftarrow Q(s_t, a_t)$ ▷ *$Q_{old} \leftarrow Q(s_t, a_{t+1})$ in the paper : error ?*
    $Q(s_t, a_t) \leftarrow R(s_t, a_t) + \gamma \sum_{s'} \frac{T(s_t, a_t, s')}{\sum_i T(s_t, a_t, i)} \max_k Q(s', k)$ ▷ *1-step MB update the Qvalue with the pseudo transition matrix T. $Q(s_t, a_{t+1})$ is updated in the paper : error ?*
    $\delta = |Q(s_t, a_t) - Q_{old}|$ ▷ *TD error to choose to add the experience in the PQueue $\delta = |Q(s_{t-1}, a_{t+1}) - Q_{old}|$ is written in the paper : error ?*
    for each (s, a) so that $T(s, a, s_t) \neq 0$ **do** ▷ *iterate on all predecessor [state-action] couple which get to the state s_t*
      $p \leftarrow \delta \times \gamma \times \frac{T(s,a,s_t)}{\sum T(s,a,i)}$ ▷ *compute the priority $p =$ error discounted by gamma and the probability of transition.*
      if $p > \nu$ **then**
        if $(s, a) \notin PQueue$ **then**
          Put (s, a) in $PQueue$ with *priority* p
        else
          Update priority of (s, a) in $PQueue$ with p
    $s_t \leftarrow s_{t+1}$
    $Q \leftarrow bidirectionalInference(Q, PQueue, s_t)$ ▷ *offline phase computed after every online step*
  until $nbActions = nbActionsMax$

Algorithm 4 Bidirectional planning

procedure BIDIRECTIONALINFERENCE($Q, PQueue, s_t$) ▷ *current state in the real world,n Q-values, priority queue*
$nbLoops \leftarrow 0$
repeat ▷ *repeat (PS/ TS) until TD error convergence or budget max*
$Sum\delta \leftarrow 0$
$nbLoops \leftarrow nbLoops + 1$
— *Start of Prioritized Sweeping (PS)* —
$nbPS \leftarrow 0$
while $priority(PQueue[0]) > \nu$ **and** $nbPS < nbPSmax$ **do** ▷ *PQueue[0] = highest prioritized tuple (s, a) / $nbPSmax$ = budget of PS*
$nbPS \leftarrow nbPS + 1$
$(s, a) \leftarrow PQueue[0]$ ▷ *(s,a) are the ones with the higher priority*
$Q_{old} \leftarrow Q(s, a)$
$Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s'} \frac{T(s,a,s')}{\sum_i T(s,a,i)} \max_k Q(s', k)$ ▷ *"$\max_k Q(s,k)$" : error in the paper ?*
$\delta \leftarrow |Q(s, a) - Q_{old}|$
$Sum\delta \leftarrow Sum\delta + \delta$
for each (s', a') so that $T(s', a', s) \neq 0$ **do** ▷ *(s', a') predecessors update priority of (s, a))*
$p \leftarrow \delta \times \gamma \times \frac{T(s,a,s_t)}{\sum T(s,a,i)}$
if $p > \nu$ **then**
if $(s', a') \notin PQueue$ **then**
Put (s', a') in $PQueue$ with *priority* p
else
Update *priority* of (s', a') in $PQueue$ with p
— *Start of Trajectory Sampling (TS)* —
$nbTS \leftarrow 0$
$s \leftarrow s_t$ ▷ *start from the current location in the real world*
while $s \notin PQueue$ **and** $nbPS < nbPSmax$ **do** ▷ *loop until a state of PQueue is reached or until budget expended*
$nbTS \leftarrow nbTS + 1$
$a \leftarrow draw(softmax_{\beta_R}(Q(s)))$ ▷ *"a" taken by the policy : maybe different from "a_t" because Q has change with PS*
$s' \leftarrow draw(probabilityProportionateSelection(T(s, a)))$ ▷ *s' is the NEXT step. And it can't be the return of the environment because we are offline. So s' is a estimation by the transitional state function. $R(s, a)$ is also the reward estimation.*
$Q_{old} \leftarrow Q(s, a)$
$Q(s, a) \leftarrow R(s, a) + \gamma \sum_{s'} \frac{T(s,a,s')}{\sum_i T(s,a,i)} \max_k Q(s', k)$ ▷ *"$\max_k Q(s,k)$" : error in the paper ?*
$Sum\delta \leftarrow Sum\delta + |Q(s, a) - Q_{old}|$ ▷ *add the error to the cumulative sum of error*
$s \leftarrow s'$ ▷ *jump to the next state ! and go on !*
until $Sum\delta < \epsilon$ or $nbLoops > nbLoopsMax$ ▷ *no significant Q-value update in the last loop or global budget exhausted*
return Q ▷ *updated Q-values*

4 Linear Successor Features Model (LSFM)

State representation is a key element of the generalization process, compressing a high-dimensional input space into a low-dimensional latent state space. Indeed, it may become computationally very difficult to compute decision-making strategies that maximize rewards for high dimensional inputs. This "curse of dimensionality" (Bellman, 1961) can be overcome by compressing high dimensional sensor data into a lower dimensional latent state space. Furthemore, mapping high-dimensional inputs into a lower dimensional latent state space can lead to faster learning because information can be reused across different inputs.

Successor features are the generalization of Successor Representation on latent space predicting frequencies of future observations. Previous work presents algorithms that reuse Successor Features (SFs) (Barreto and Dabney, 2017) to initialize learning across tasks with different reward specifications, leading to improvements in learning speed in challenging control tasks (Barreto and Borsa, 2019); (Zhang, 2016); (Gershman, 2016). Nevertheless, the latent space learned by these Successor Features cannot generalize across different transition functions.

Here, we introduce the Linear Successor Feature Model (LSFM) (Lehnert, 2020). LSFMs construct latent state space that extract equivalences of a task's transition dynamics and rewards, and afford transfer across tasks with different rewards but also different transition dynamics (unlike previous SFs). The extension from (Lehnert, 2020) of this present work is to learn a neural networks mapping inputs to latent feature spaces that are predictive of future reward outcomes. It is presented in section 4.5.

4.1 State Representations

4.1.1 Value-predictive state representation

A latent state space S_ϕ is constructed using a state representation function $\phi : S \to S_\phi$. A value-predictive state representation constructs a latent state space that retains enough information to support accurate value or Q-value predictions. Authors in (Barreto and Dabney, 2017) use directly SFs as state representation, the SFs formulation is value-predictive :

$$Q^\pi(s,a) = \boldsymbol{\psi}^\pi(s,a).\boldsymbol{w} \tag{39}$$

Because value-predictive state representations are only required to be predictive of a particular policy π, they implicitly depend on the policy π. Consequently, this formulation of SFs learns latent state space that is predictive of the optimal decision-making strategy and are thus akin to model-free RL.

4.1.2 Reward-Predictive State Representations

A reward-predictive state representation constructs a latent state space that retains enough information about the original state space to support accurate

predictions of future expected reward outcomes. Which means for any start state and any arbitrary action sequence, both the latent grid world and the original grid world produce equal reward sequences. This property can be formalized using a family of functions $\{f_t\}_{t\in\mathbb{N}}$ that predicts the expected reward outcome after executing an action sequence $a_1, ..., a_t$ starting at state s:

$$f_t : (s, a_1, ..., a_t) = \mathbb{E}_p[r_t|s, a_1, ..., a_t] \tag{40}$$

A state representation is *reward-predictive* if the function f_t can be reparameterized in terms of the constructed latent state space and if there exists a family of functions $\{g_t\}_{t\in\mathbb{N}}$ such that

$$f_t : (s, a_1, ..., a_t) = g_t(\phi(s), a_1, ..., a_t) \tag{41}$$

4.2 Successor Features (SFs)

SFs combine Successor Representation with arbitrary state representations. Given a state representation ϕ, the SF is a column vector defined for each state and action pair (Gershman, 2016) and

$$\boldsymbol{\psi}^{\pi}(s, a) = \mathbb{E}_{p,\pi}\left[\sum_{t=1}^{\infty} \gamma^{t-1}\boldsymbol{\phi}_{st}\middle| s_1 = s, a_1 = a\right], \tag{42}$$

A SF vector $\boldsymbol{\psi}^{\pi}$ can be understood as a statistic measuring how frequently different latent states vectors are encountered following the policy π.

Suppose a state representation ϕ constructs a latent state space with empirical latent transition probabilities that match the transition probabilities of the original task, this state representation is reward predictive. A reward-predictive state representation can be used in conjunction with a Linear Action Model (Sutton, 2008) to compute expected future reward outcomes.

Definition 1 — *Linear Action Model (LAM) : Given an MDP and a state representation $\phi : \mathcal{S} \to \mathbb{R}^n$ a LAM consists of a set of matrices and vectors $\{\boldsymbol{M}_a, \boldsymbol{w}_a\}_{a\in\mathcal{A}}$, where $\boldsymbol{M}_a$ is of size $n \times n$ and the column vector $\boldsymbol{w}_a$ is of dimension n.*

The transition matrices of a LAM $\{\boldsymbol{M}_a\}_{a\in\mathcal{A}}$ model the empirical latent transition probabilities. The vectors $\boldsymbol{w}_a$ model a linear map from latent states to expected one-step reward outcomes. The expected reward outcome after following the action sequence $a_1, ..., a_t$ starting at state s can then be approximated with

$$\mathbb{E}_p[r_t|s, a_1, ..., a_t] \approx \boldsymbol{\phi}_s^{\top}\boldsymbol{M}_{a1}...\boldsymbol{M}_{at-1}.\boldsymbol{w}_{at} \tag{43}$$

To tie SFs to reward-predictive state representations, we first introduce a set of square real-valued matrice $\{\boldsymbol{F}_a\}_{a\in\mathcal{A}}$ such that, for every state s and action a,

$$\boldsymbol{\phi}_s^{\top}\boldsymbol{F}_a \approx \boldsymbol{\psi}^{\pi}(s, a), \tag{44}$$

where π is defined on the latent state space. A Linear Successor Feature Model (LSFM) is then defined using the matrices $\{\boldsymbol{F}_a\}_{a\in\mathcal{A}}$:

Definition 2 — *Linear Successor Feature Model (LSFM) : Given an MDP, a policy π, and a state representation $\phi : \mathcal{S} \to \mathbb{R}^n$, an LSFM consists of a set of matrices and vectors $\{\boldsymbol{F}_a\}_{a\in\mathcal{A}}$, where $\boldsymbol{F}_a$ is of size $n \times n$ and the column vector $\boldsymbol{w}_a$ is of dimension n. The $\{\boldsymbol{F}_a\}_{a\in\mathcal{A}}$ are used to model a linear map from latent state features to SFs as described in eq.* (44).

In (Barreto and Dabney, 2017), the latent space is constructed by setting the output of $\boldsymbol{\phi}$ to be one-step rewards : $\boldsymbol{r}(s, a, s') = \boldsymbol{\phi}(s, a, s').\boldsymbol{w}$. In constrast, LSFMs are used to learn the state-representation function ϕ that satisfies eq. (44) suitable for predicting reward sequences for any arbitrary decision sequence. Thus, LSFM reward-predictive state representations can be understood as form of model-based RL. With LSFM, an agent has to search policies that are defined on its latent state space. These policies are called *abstract policies*.

Definition 3 — *Abstract Policies : An abstract policy π_ϕ is a function mapping latent state and action pairs to probability values:*

$$\forall s \in \mathcal{S}, a \in \mathcal{A}, \pi_\phi(\boldsymbol{\phi}_s, a) \in [0, 1] \text{ and } \sum_a \pi_\phi(\boldsymbol{\phi}_s, a) = 1 \tag{45}$$

For a fixed state representation ϕ, the set of all abstract policies is denoted with Π_ϕ.

4.3 LSFM

4.3.1 Bisimulation

Bisimulation Relations express the fact that the empirical latent transition probabilities have to match the transition probabilities in the original task. In (Lehnert, 2020), the author proves that LAMs or LSFMs encode state representations that generalize only across bisimilar states. This affirmation is based on the assumption that the state representation $\phi : \mathcal{S} \to \{\boldsymbol{e}_1, ..., \boldsymbol{e}_n\}$ is assumed to have a range that consists of all n one-hot bit vectors $\phi(s) = \boldsymbol{e}_i$.

Theorem 1 — *LSFM one-hot latent states : For an MDP $\langle \mathcal{S}, \mathcal{A}, p, r, \gamma \rangle$ be a state representation and $\{\boldsymbol{F}_a, \boldsymbol{w}_a\}_{a\in\mathcal{A}}$ an LSFM. If, for one policy $\pi \in \Pi_\phi$, the representation ϕ satisfies*

$$\forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \boldsymbol{\phi}_s^\top.\boldsymbol{w}_a = \mathbb{E}_p[r(s, a, s')|s, a] \text{ and } \boldsymbol{\phi}_s^\top \boldsymbol{F}_a = \boldsymbol{\phi}_s^\top + \gamma \mathbb{E}_{p,\pi}[\boldsymbol{\phi}_{s'} \boldsymbol{F}_{a'}|s, a], \tag{46}$$

then ϕ generalizes across bisimilar states and any two states s and $\tilde{s}$ are bisimilar if $\boldsymbol{\phi}_s = \boldsymbol{\phi}_{\tilde{s}}$. If eq. (46) *holds for one policy $\pi \in \Pi_\phi$, then eq.* (46) *also holds every other policy $\tilde{\pi} \in \Pi_\phi$ as well.*

But the assumption of one-hot bit vectors can be relaxed by a state representation which only approximately satisfies the conditions outlined in theorem 1. Moreover, an (approximate) reward-predictive state representation (approximately) generalizes across all abstract policies, because the same state representation can be used to predict the value of every possible abstract policy $\pi \in \Pi_\phi$ if prediction errors are low enough.

4.3.2 Learning

To learn the (approximate) reward-predictive state representation with LSFM, the loss objective $\mathcal{L}_{LSFM}$ is the sum of three different terms: The first term $\mathcal{L}_r$ computes the one-step reward prediction error and is designed to minimize the reward error ϵ_r. The second term $\mathcal{L}_\psi$ computes the SF prediction error and is designed to minimize the SF error ϵ_ψ. The last term $\mathcal{L}_N$ is a regularizer constraining the gradient optimizer to find a state representation that outputs unit norm vectors.

Given a finite data set of transitions $D = (s_i, a_i, r_i, s'_i)_{i=1}^D$, the formal loss objective is

$$\mathcal{L}_{LSFM} = \underbrace{\sum_{i=1}^{D} \left(\boldsymbol{\phi}_{s_i}^\top \boldsymbol{w}_{a_i} - r_i\right)^2}_{=\mathcal{L}_r} + \alpha_\psi \underbrace{\sum_{i=1}^{D} \left\| \boldsymbol{\phi}_{s_i}^\top \boldsymbol{F}_a - \vec{\boldsymbol{y}}_{s_i,a_i,r_i,s'_i} \right\|_2^2}_{=\mathcal{L}_\psi} + \underbrace{\alpha_N \sum_{i=1}^{D} \left(\left\| \boldsymbol{\phi}_{s_i} \right\|_2^2 - 1 \right)^2}_{=\mathcal{L}_N} \tag{47}$$

and α_N, $\alpha_\psi > 0$ are hyper-parameters. In eq. (47), the prediction target is

$$\vec{\boldsymbol{y}}_{s,a,r,s'} = \boldsymbol{\phi}_s^\top + \gamma \boldsymbol{\phi}_{s'}^\top \overline{\boldsymbol{F}} \tag{48}$$

with $\overline{\boldsymbol{F}}$ the LSFM that predicts the SF for a policy that selects actions **uniformly at random** :

$$\overline{\boldsymbol{F}} = \frac{1}{|\mathcal{A}|} \sum_{a \in \mathcal{A}} \boldsymbol{F}_a \tag{49}$$

Because the matrix $\overline{\boldsymbol{F}}$ averages across all actions, the LSFM computes the SFs for a policy that selects actions uniformly at random. But the matrix $\overline{\boldsymbol{F}}$ could be constructed differently because the proofs of the theoretical results assume that $\overline{\boldsymbol{F}}$ can only depend on the matrices $\{\boldsymbol{F}_a\}_{a \in \mathcal{A}}$ and is not a function of the state s.

For each gradient update, the target $\vec{\boldsymbol{y}}_{s,a,r,s'}$ can be considered a constant (Lehnert, 2020).

4.3.3 Results from (Lehnert, 2020)

Figure 5 presents the puddle-world experiments and the results. To analyze across which states the learned reward-predictive state representation generalizes, all feature vectors were clustered using agglomerative clustering. Two different states that are associated if the distance of there feature vectors $\|\boldsymbol{\phi}_s - \boldsymbol{\phi}_{\tilde{s}}\|^2$

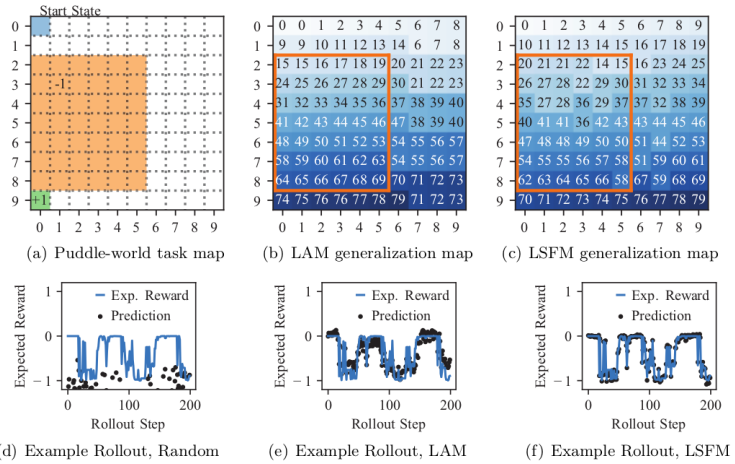


Figure 5: Puddle-World Experiment from (Lehnert, 2020).
6(a): Map of the puddle-world task in which the agent can move up, down, left, or right to transition to adjacent grid cells. The agent always starts at the blue start cell and once the green reward cell is reached a reward of +1 is given and the interaction sequence is terminated. For each transition that enters the orange puddle, a reward of -1 is received. 6(b), 6(c): Partitioning obtained by merging latent states into clusters by Euclidean distance. 6(d), 6(e), 6(f): Expected reward and predictions for a randomly chosen 200-step action sequence using a randomly chosen representation, a representation learned with a LAM, and a representation learned with an LSFM.

is low. Figure 5 (b) and (c) plot the obtained clustering as a partition map. Only a transition data set $\mathcal{D}$ was given as input to the optimization algorithm and the algorithm was not informed about the grid-world topology of the task in any other way.

Figure 5 (d), (e), (f) plot an expected reward rollout and the predictions presented by a random initialization (Figure 5 (d)), the learned representation using a LAM (Figure 5 (e)), and the learned representation using a LSFM (Figure 5 (f)). The blue curve plots the expected reward outcome $\mathbb{E}_p[r_t|s, a_1, ..., a_t]$ as a function of t for a randomly selected action sequence. While a randomly initialized state representation produces poor predictions of future reward outcomes (Figure 5 (d)), the learned representations produce relatively accurate predictions and follow the expected reward curve (Figure 5 (e) and (f)). Because the optimization process was forced to compress 100 grid cells into a 80 dimensional latent state space, the latent state space cannot preserve the exact grid cell position and thus approximation errors occur.

The plots in Figure 5 suggest that both LSFMs and LAMs can be used to learn approximate reward-predictive state representations. Because both models optimize different non-linear and non-convex loss functions, the optimization

process leads to different local optima, leading to different performance on the puddle-world task. While prediction errors are present, Figure 5 suggests that both LSFM and LAM learn an approximate reward-predictive state representation and empirically the differences between each model are not significant.

4.4 Discussion on LSFM

The schematic in Figure 6 illustrates the differences between the previous and the presented models. LSFM ties successor features to model-based reinforce-

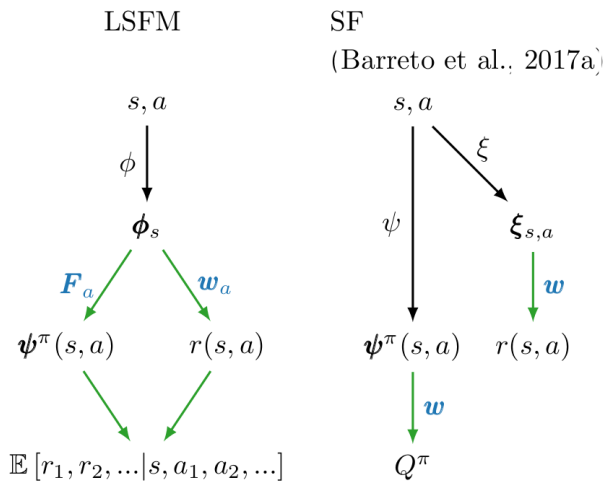


Figure 6: Comparison of SFs State-Representation Models

ment learning, because an agent that has learned an LSFM can predict expected future reward outcomes for any arbitrary action sequence. LSFMs are a "strict" model-based architecture and are distinct from model-based and model-free hybrid architectures that iteratively search for an optimal policy and adjust their internal representation.

LSFMs only evaluate SFs for a fixed target policy that selects actions uniformly at random. The learned model can then be used to predict the value function of any arbitrary policy, including the optimal policy.

In contrast to the SF framework introduced by (Barreto and Dabney, 2017), the connection between LSFMs and model-based RL is possible because the same state representation ϕ is used to predict its own SF (Figure 6). While the deep learning models presented by (Gershman, 2016) and (Zhang, 2016) use one state representation to predict SFs and one-step rewards, these models are also constrained to predict image frames. LSFMs do not use the state representation to reconstruct actual states. Instead, the state space is explicitly compressed, allowing the agent to generalize across distinct states. LSFMs do not use the state representation to reconstruct actual states. Instead, the state space is

explicitly compressed, allowing the agent to generalize across distinct states.

Reward-predictive state representations remove previous limitations of SFs and generalize across variations in transition functions. This property stands in contrast to previous work on SFs, which demonstrate robustness against changes in the reward function only ((Barreto and Dabney, 2017), (Barreto and Borsa, 2019), (Gershman, 2016), (Zhang, 2016)).

In all cases, including reward-predictive state representations, the learned models can only generalize to changes that approximately preserve the latent state space structure. If the same representation is used on a completely different randomly generated finite MDP, then positive transfer may not be possible because both tasks do not have a latent state structure in common.

The combination of LSFM with efficient exploration algorithms is left to future studies.

4.5 Linear Successor Feature Neural Network (LSFNN)

To exploit the potentiality of LSFM in bigger state space, we construct here a Linear Successor Feature Neural Network (LSFNN). But in a first step, we test the LSFNN in simple environment of navigation tasks.

The first experiment is made on a small 6×6 puddle word described in Figure 7a. The purpose is to compare the cumulative rewards of a linear Q model trained with a reward predictive latent space with a Deep Q Learning model.

The second experiment involve two bigger environments. The first one is a bigger puddle word of 11×11. A reward predictive latent space and a value predictive latent space are constructed on this first environment. These two latent space are then reused to train a linear Q-learning on the second four-rooms environment, described in fig. 10b which differs on rewards and transition.

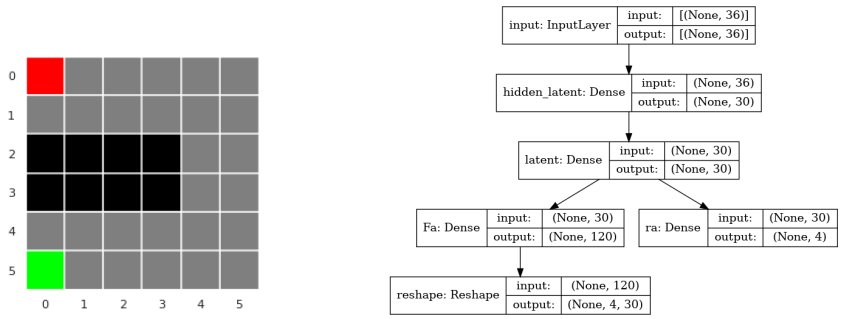
4.5.1 Small puddle word

The environment is a small puddle word on a 6×6 grid presented in Figure 7a. The environment initial state is fixed as the goal state. The Black states return rewards of -1, grey states return rewards of 0.

The architecture of LSFNN presented in Figure 7b contain 30 hidden layers and constructs a 30 dimensional latent state space. The linear reward prediction layer $\boldsymbol{w}_a$ such as the output is $\hat{\boldsymbol{r}}(\boldsymbol{\phi}_s, a) = \boldsymbol{\phi}_s^\top \boldsymbol{w}_a$, and the linear matrix $\{\boldsymbol{F}_a\}$ such as the output is $\hat{\boldsymbol{\psi}}(\boldsymbol{\phi}_s, a) = \boldsymbol{\phi}_s^\top \boldsymbol{F}_a$ are dense layers with no bias.

LSFNN is trained on a fixed target policy that selects actions uniformly at random. The loss is exactly calculated following eq. (47). After trained on 200 episodes, the LSFNN losses converge, as presented in the Figure 8.

As the selection of actions is uniformly random, the agent reach the goal on 150 steps in average, so the whole simulation takes 30000 steps. The loss on $\boldsymbol{\psi}$ is the slower because $\alpha_\psi = 0.01$ and $\alpha_N = 1$. These coefficients were obtained by fine tuning with $\alpha_\psi \in [0.01, 0.1, 1]$ and $\alpha_N \in [0.01, 0.1, 1]$.



(a) Simple puddle word 6×6 (b) LSFNN Architecture

Figure 7: (a) Simple 6×6 puddle word with fix red initial state and green goal state . Black states return rewards of -1, grey states return rewards of 0. The green goal states return a $+1$ reward. (b) LSFNN architecture with a 30 latent space dimension. The Linear reward prediction layer $\{\boldsymbol{w}_a\}$ and the linear $\{\boldsymbol{F}_a\}$ matrix are dense layer with no bias.

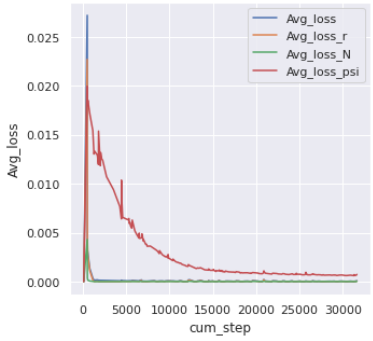


Figure 8: Losses on LSFNN

The learned latent space $\boldsymbol{\phi}_s$ constructed by LSFNN can be used to predict the valuefunction of any arbitrary policy, including the optimal policy. We trained a Linear Q-learning Neural Network (QNN) on the latent space $\boldsymbol{\phi}_s$ and compared with a Linear QNN and a Deep Q-learning (DQN) with 30 hidden layers on the original state space.

The experiment is repeated 10 times for each of the 3 configurations trained on 200 episodes. The policy is an espilon-greedy with a exponential decrease of epsilon with the number of steps : $\epsilon = \epsilon_{min} + (\epsilon_{max} - \epsilon_{min}) * \exp(-\lambda steps)$, with $\epsilon_{min} = 0.01$, $\epsilon_{max} = 1$, and $\lambda = 0.0005$. The results in cumulative gain is presented on the Figure 9.

The drop of cumulative results on the first episodes mean the models gather lots of negative rewards in the beginning before starting to learn to avoid them.

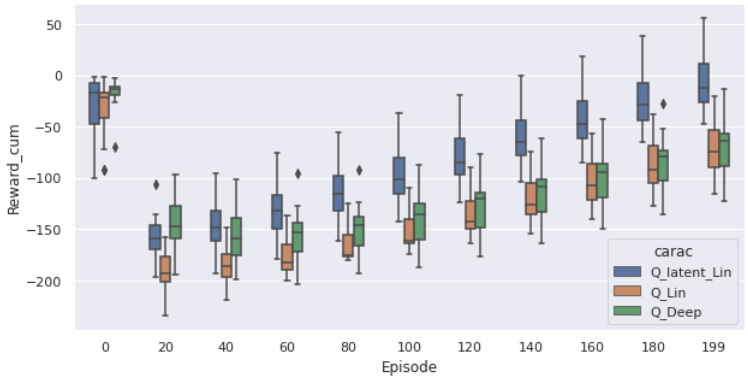


Figure 9: Q learnings on latent and original state space (linear and Deep)

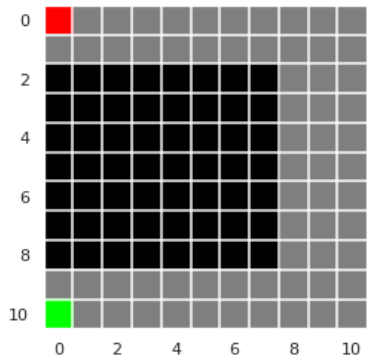
The results show clearly the best results on the linear Q learning based on latent space trained with LSFNN.

4.5.2 LSFNN vs DQN latent spaces on two environments

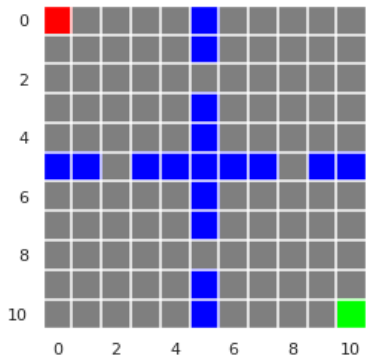
The goal of this section is to confirm experimentally the potentiality of LFSNN to transfer tasks which differ in rewards and transitions.

We first construct a reward predictive latent space with LFSNN on a puddle word of 11×11 as described in Figure 10a. The policy and the model parameters to train LSFNN are the same as described in Section 4.5.1.

A value predictive latent space is constructed with a DQN with 30 hidden layers on the original state space. The only change in comparison with the DQN training in section 4.5.1, is the decrease of $\lambda = 0.0001$ for the exponential decrease of epsilon greedy.



(a) Big puddle word 11×11



(b) 11×11 Four rooms environment

Figure 10: (a) Big 11×11 puddle word with fix red initial state and green goal state. Black states return rewards of -1, grey states return rewards of 0. The green goal states return a $+1$ reward. (b) 11×11 Four rooms environment with fix red initial state and green goal state. Blue tiles are barriers where the agent can not go through and grey states return rewards of 0.

These two latent space are then reused to train a linear Q-learning on a second four-rooms environment described in Figure 10b which differs on rewards and transition.

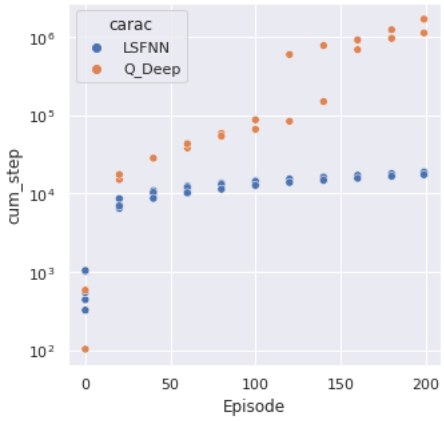


Figure 11: Cumulative steps of a linear Q-learning model, trained with latent space constructed on Big puddle word by LSFNN and DQN model.

We compare in Figure 11 the cumulative steps of a linear Q-learning model, trained with latent space constructed on Big puddle word by LSFNN and DQN model. It is clear that latent space constructed by the reward predictive LSFNN

model on the Big puddle word leads to way better results for the learning on the 4-rooms environment, than the value predictive DQN latent space.

We confirm experimentally the capacity of the reward predicitve LSFNN model to transfer representations across different tasks which differ in both the rewards and transitions.

4.5.3 Future work

The training time of LSFNN is very long with a uniform random policy. A idea of future work is to improve the training time of LSFNN by finding a better policy than the selection of actions uniformly at random.

A promising way is the *option* model presented by (Machado, 2018.).

References

A. Barreto and D. Borsa. *Transfer in Deep Reinforcement Learning Using Successor Features and Generalised Policy Improvement.* 2019.

A. Barreto and W. Dabney. *Successor Features for Transfer in Reinforcement Learning.* 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA., 2017.

Cazé-Khamassi. *Hippocampal replays under the scrutiny of reinforcement learning models.* J Neurophysiol, 2018.

P. Dayan. *Improving Generalization for Temporal Difference Learning: The Successor Representation.* Computational Neurobiology Laboratory, The Salk Institute, P.0. Box 85800, Sun Diego, CA 92186-5800 USA, 1993.

S. J. Gershman. *The Successor Representation: Its Computational Logic and Neural Substrates.* The Journal of Neuroscience, 2018. ISBN 9781417642595.

T. D. K. S. J. Gershman. *Deep Successor Reinforcement Learning.* ArXiv vol abs/1606.02396, 2016.

Khamassi-Girard. *Modeling awake hippocampal reactivations with model-based bidirectional search.* Biological Cybernetics (Modeling), Springer Verlag, 2020, 10.1007/s00422-020- 00817-x. hal-02504897, 2020.

L. Lehnert. *Successor Features Combine Elements of Model-Free and Model-based Reinforcement Learning.* 2020.

Lin. *Self-improving reactive agents based on reinforcement learning, planning and teaching.* Machine learning, 8(3):293–321, 1992, 1992.

Machado. *EIGENOPTION DISCOVERY THROUGH THE DEEP SUCCESSOR REPRESENTATION.* conference paper at ICLR 2018, 2018.

Mattar-Daw. *Prioritized memory access explains planning and hippocampal replay.* Nature Neuroscience, 2018.

Mnih. *Playing atari with deep reinforcement learning.* ICLR 2016arXiv:1312.5602, 2013.Mnih, 2013.

Mnih. *Human-level control through deep reinforcement learning.* Nature, 518(7540):529–533, 2015, 2015.

Schaul. *Prioritized experience replay.* ICLR, 2016.

Sutton. *Integrated Modeling and Control Based on Reinforcement Learning and Dynamic Programming.* nips, 1990.

R. S. Sutton. *Dyna-style planning with linear function approximation and prioritized sweeping.* 24th Conference on Uncertainty in Artificial Intelligence, 2008.

Sutton-Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998 A Bradford Book, 1998.

J. Zhang. *Deep Reinforcement Learning with Successor Features for Navigation across Similar Environments.* 2016.