

# **FLoc / FoPPS Summer School**

## **Temporal Abstraction**

Doina Precup

McGill University & Google DeepMind Montreal

[dprecup@cs.mcgill.ca](mailto:dprecup@cs.mcgill.ca)

With thanks to Rich Sutton, Satinder Singh, Pierre-Luc Bacon, Jean Harb

# What is temporal abstraction?

- Consider an activity such as cooking dinner



- High-level steps: choose a recipe, make a grocery list, get groceries, cook,...
- Medium-level steps: get a pot, put ingredients in the pot, stir until smooth, check the recipe ...
- Low-level steps: wrist and arm movement while driving the car, stirring, ...
- All have to be seamlessly integrated!
- Cf. macro actions in classical AI, controllers in robotics

# Why is temporal abstraction useful for complex tasks

- *Advantages to planning*
  - Need to generate shorter plans
  - Improves robustness to model errors
  - Might need to look at fewer states, since the abstract actions have pre-defined termination conditions
  - Discretize the action space in continuous problems
- *Advantages to learning*
  - Improves exploration (can travel in larger leaps)
  - Gives a natural way of using a single stream of data to learn many things
- *Advantages to interpretability:*
  - Focusing attention: Sub-plans ignore a lot of information
  - Improves readability of both models and resulting plans
  - Reduces the problem size

# Formalization of temporal abstraction

- Hierarchical abstract machines (Parr, 1998)
- MAXQ (Dietterich, 1998)
- Dynamic motion primitives (Schaal et al. 2004)
- Skills (Konidaris et al, 2009)
- *Options* (Sutton, Precup & Singh, 1999; Precup, 2000)

# Options framework

- An *option*  $o$  consists of 3 components
  - An *initiation set* of states  $I_o \subseteq S$  (aka precondition)
  - A *policy*  $\pi_o : S \times A \rightarrow [0, 1]$   
 $\pi_o(s, a)$  is the probability of taking  $a$  in  $s$  when following the option
  - A *termination condition*  $\beta_o : S \rightarrow [0, 1]$ :  
 $\beta(s)$  is the probability of terminating the option at  $s$
- Eg., robot navigation: if there is no obstacle in front ( $I_o$ ), go forward ( $\pi_o$ ) until you get too close to another object ( $\beta_o$ )
- Other representations of the termination condition are possible (cf. Comanici and Precup, 2010)

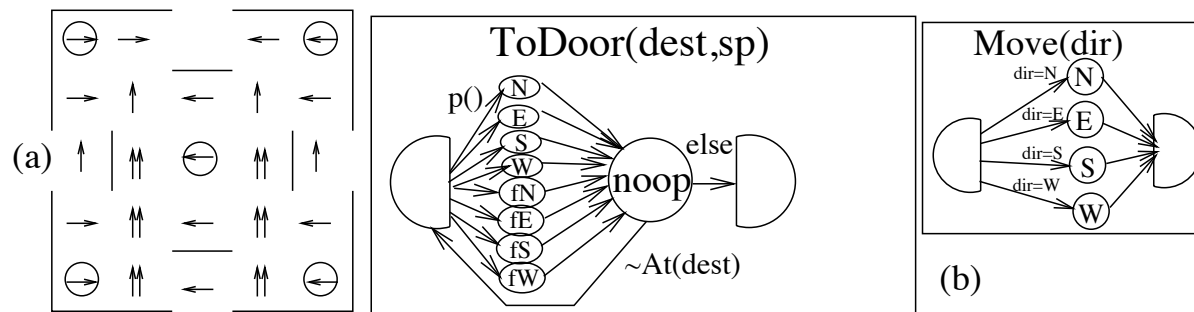
Cf. Sutton, Precup & Singh, 1999; Precup, 2000

# Options as behavioral programs

- *Call-and-return execution*
  - Option is a subroutine which gets called by a policy over options  $\pi_{\Omega}$
  - When called,  $\omega$  is pushed onto the execution stack
  - During the option execution, the program looks at certain *variables (aka state)* and executes an *instruction (aka action)* until a termination condition is reached
  - The option can keep track of additional *local variables*, eg counting number of steps, saturation in certain features (e.g. Comanici, 2010)
  - *Options can invoke other options*
- *Interruption*
  - At each step, one can check if a better alternative has become available
  - If so, the option currently executing is *interrupted* (special form of concurrency)
- *The option identity is also a form of memory: what is the agent currently trying to achieve?* Cf. Shaul et al, 2014, Kulkarni et al, 2016

# Alternative formalisms

- MAXQ (Dietterich, 2000): specify a graph of sub-tasks
- Hierarchies of Abstract Machines (Parr & Russell, 1997): abstractions given by automata
- Andre (2003): programming language (ALISP) which allows specifying HAM programs, learning parts of them



# Option models

- *Option model* has two parts:
  1. *Expected reward*  $r_\omega(s)$ : the expected return during  $\omega$ 's execution from  $s$ 
    - Needed because it is used to update the agent's internal representations
  2. *Transition model*  $P_\omega(s'|s)$ : a sub-probability distribution over next states (reflecting the discount factor  $\gamma$  and the option duration) given that  $\omega$  executes from  $s$ 
    - $P$  specifies *where* the agent will end up after the option/program execution and *when* termination will happen
- Models are *predictions* about the future, conditioned on the option being executed



## Option models provide semantics

- Programming languages: preconditions (initiation set) and postconditions
- *Models of options represent (probabilistic) post-conditions*
- *Models that are compositional*, can be used to reason about the policy over options
- *Sequencing*

$$\mathbf{r}_{\omega_1\omega_2} = \mathbf{r}_{\omega_1} + P_{\omega_1}\mathbf{r}_{\omega_2}$$

$$P_{\omega_1\omega_2} = P_{\omega_1}P_{\omega_2}$$

Cf. Sutton et al, 1999, Sorg & Singh, 2010

- In homogeneous coordinates:

$$\mathbf{M}_{\omega_1\omega_2} = \mathbf{M}_{\omega_1}\mathbf{M}_{\omega_2}$$

## Option models provide semantics

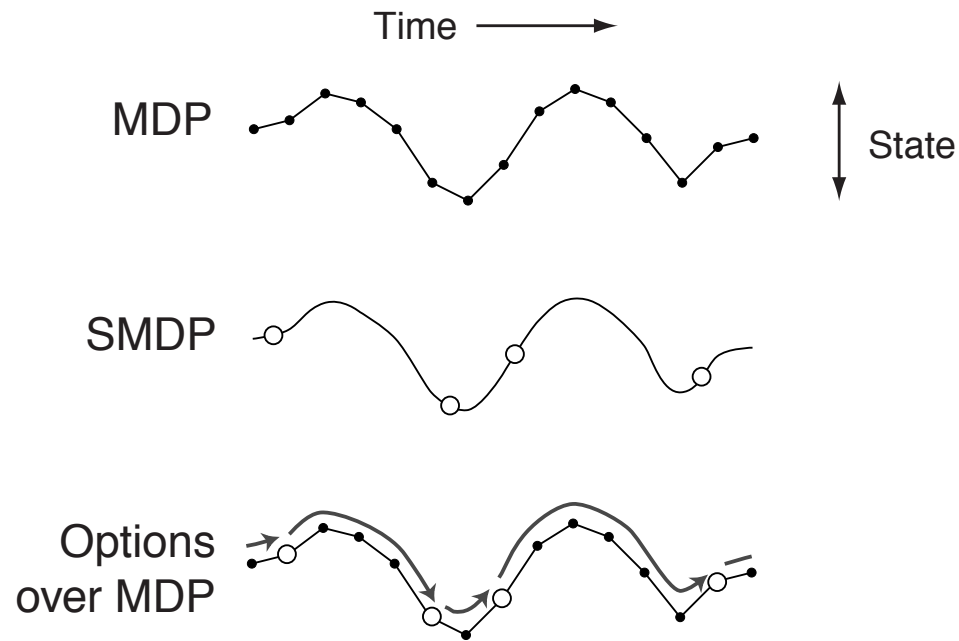
- *Stochastic choice*: can take expectations of reward and transition models, eg if we have a policy over options  $\Omega$ , its immediate reward is obtained as:

$$\sum_{\omega} \pi_{\Omega}(\omega|s) \mathbf{r}_{\omega}(s)$$

- Easy to write this in homogeneous coordinates as well.
- Convergence of Bellman equations follows immediately!
- Silver & Ciosek (2012): re-write model in homogeneous, compose models to construct programs

Eg. good generalization in Towers of Hanoi

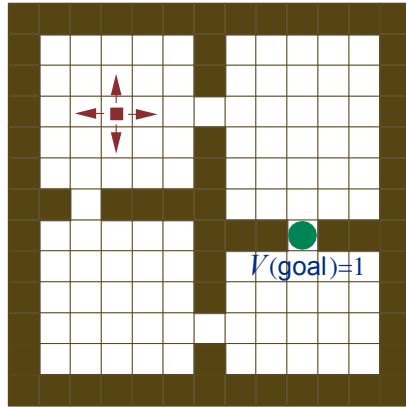
# MDP + Options = Semi-Markov Decision Process



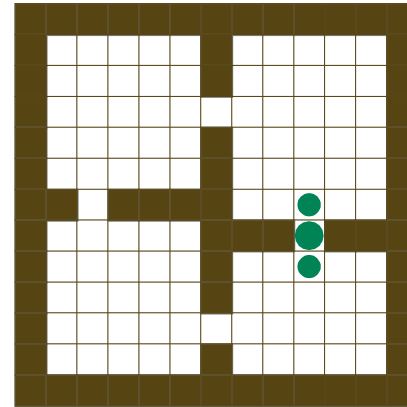
- Introducing options in an MDP induces a related semi-MDP
- Hence *all planning and learning algorithms* from classical MDPs transfer directly to options (Cf. Sutton, Precup & Singh, 1999; Precup, 2000)
- But planning and learning with options can be much faster!

# Illustration: Navigation

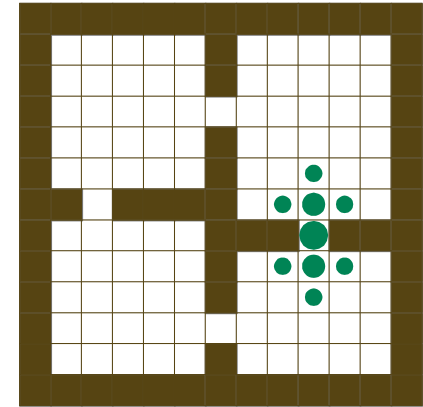
with cell-to-cell primitive actions



Iteration #0

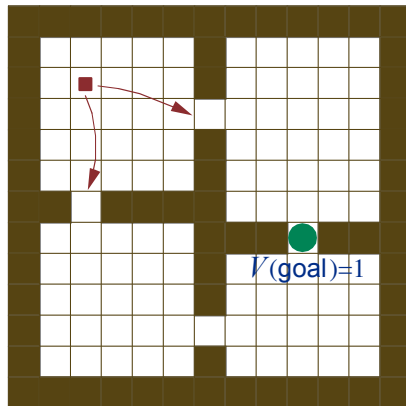


Iteration #1

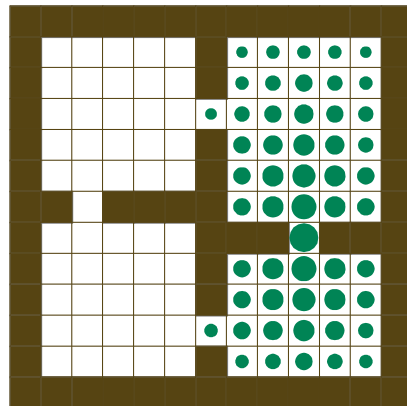


Iteration #2

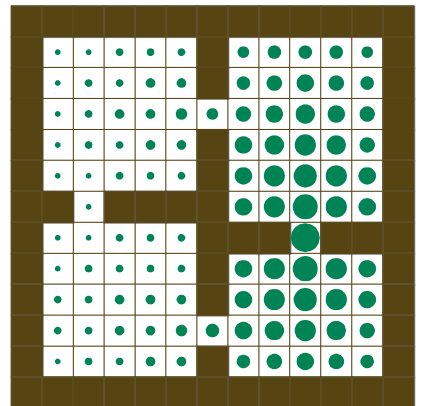
with room-to-room options



Iteration #0



Iteration #1



Iteration #2

## Advantages

- Easy to learn using temporal-difference-style methods, from a single stream of experience
- Planning with option models is done just like planning with primitives - *no explicit hierarchy*
- Result of planning with a set of options  $O$  is an option-value function, e.g.  $V_O$

## TD at the SMDP level

- Let  $N_t$  now be a random variable for the duration of an option started at time  $t$ :

$$G_t^{(N_t)} = R_t + \gamma R_{t+1} + \dots + \gamma^{N_t-1} R_{t+N_t-1} + \gamma^{N_t} Q(S_{t+N_t}, W_{t+N_t})$$

- It's just like using the  $n$ -steps return but where  $n$  is a random variable rather than being fixed.
- SMDP TD Prediction (for option values)

$$Q_{t+N_t}(S_t, W_t) = Q_{t+N_t-1}(S_t, W_t) + \alpha \left( G_t^{(N_t)} - Q_{t+N_t-1}(S_t, W_t) \right)$$

- How do you get SMDP Q-learning ? Just take a  $\max_w$  in the  $G_t^{(N_t)}$  update target.

# The anatomy of the reward option model

- Primitive action model:  $r_a(s) = \mathbb{E}[r_t | s_t = s, a_t = a]$
- Option model:

$$r_\omega(s) = \mathbb{E}[r_t + \gamma r_{t+1} + \dots | s_t = s, \omega_t = \omega]$$

- This expectation indicates a Markov-style property, as it depends only on the identity of the state and the option, not on the time step
- Notice the *model is basically a value function* so we can write Bellman equations for the model:

$$r_\omega(s) = \sum_a \pi_\omega(s, a) [r(s, a) + \sum_{s'} \gamma(1 - \beta_\omega) r_\omega(s')]$$

- This means that we can use RL methods to learn the models of options!
- Very similar equations hold for the transition model

## Intra-option algorithms

- Learning about one option at a time is very inefficient
- In fact, we may not want to execute options at all!
- Instead, *learn about all options consistent with the behaviour*
- In some sense, a form of *attention*
- E.g. action-value function, tabular case

On single-step transition  $\langle s, a, r, s' \rangle$ , for all  $\omega$  that could have been executing in  $s$  and taken  $a$ :

$$Q_{\Omega}(s, \omega) = Q_{\Omega}(s, \omega) + \alpha[r_a(s) + \gamma(1 - \beta_{\omega}(s'))Q_{\Omega}(s', \omega) + \beta_{\omega}(s')\gamma \sum_{s'} \max_{\omega'} Q_{\Omega}(s', \omega') - Q_{\Omega}(s, \omega)]$$

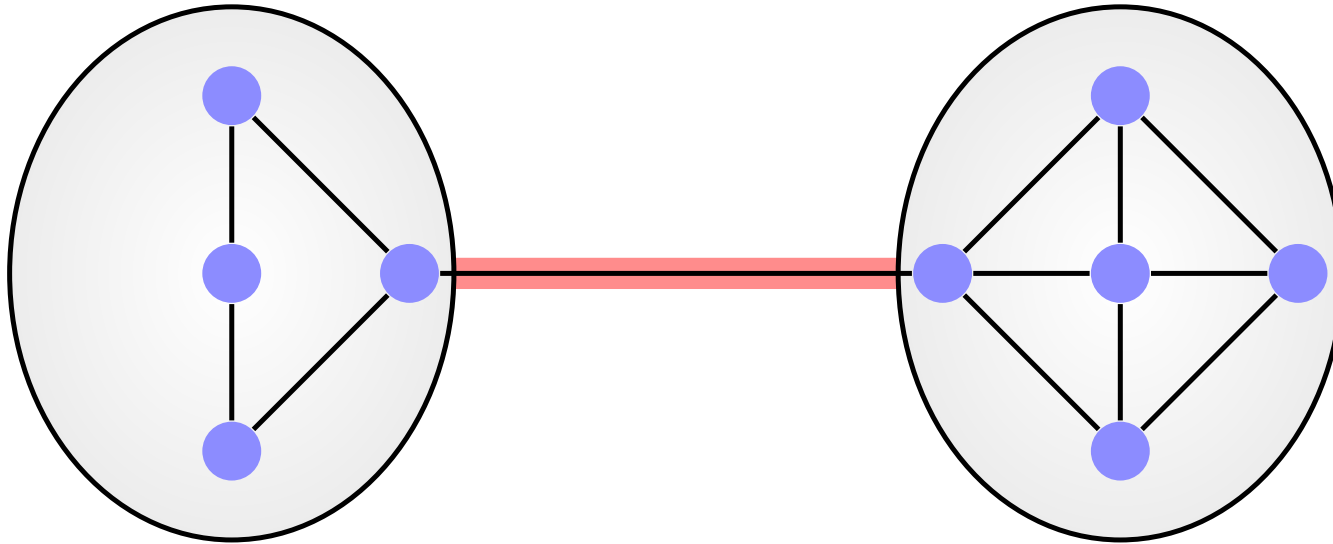
- In general function approximation, importance sampling will need to be used (several papers on this)



## Frontier: Option discovery

- Options can be given by a system designer
- If subgoals / secondary reward structure is given, the option policy can be obtained, by solving a smaller planning or learning problem (cf. [Precup, 2000](#))
- *What is a good set of subgoals / options?*
- This is a *representation discovery* problem

## Bottleneck states



1

- Perhaps the most explored idea in options construction (e.g. McGovern & Barto, 2002, Simsek & Barto, 2004)
- A bottleneck is a special state, which is visited more often than others, allows “circulating on the graph”

---

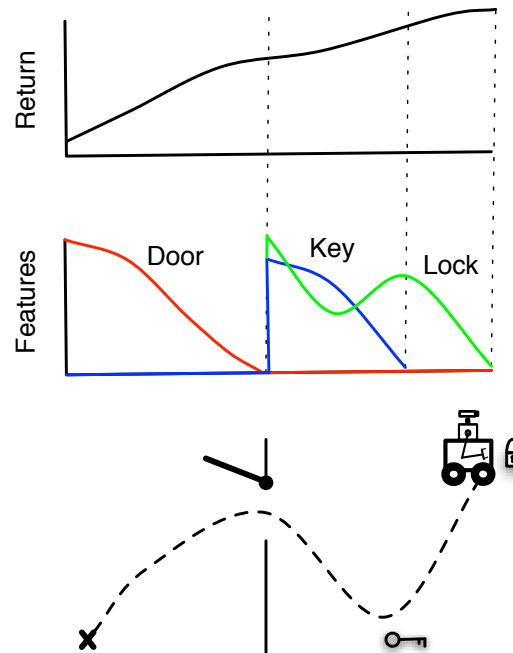
<sup>1</sup>Botvinick, 2011

## Finding bottleneck states

- Lots of different approaches!
  - Frequency of states (McGovern et al, 2001, [Stolle & Precup, 2002](#))
  - Graph partitioning / state graph analysis (Simsek et al, 2004, Menache et al, 2004, [Bacon & Precup, 2013](#))
  - Information-theoretic ideas (Peters et al., 2010)
- People seem quite good at generating these (cf. Solway et al, 2014)
- Laplace options (Machado et al, 2017)
- *Main drawback: expensive both in terms of sample size and computation*

# Change-point detection

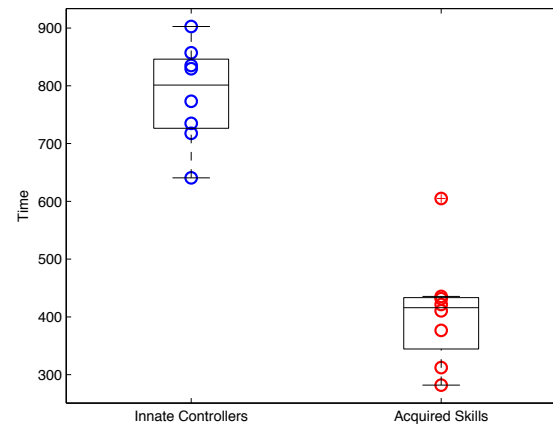
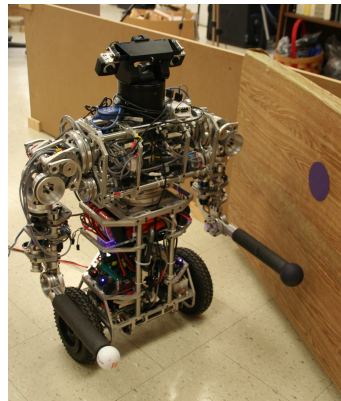
- Main idea: detect statistical changes in the time series generated by the behaviour, use this to segment into trajectories that are assumed to correspond to different options.



Cf. Konidaris et al., AAI 2012

## Applications: Robot control

- Controllers are learned automatically, at the same time as solving the task, using state-of-art Bayesian change point detection methods (Konidaris et al, 2009, 2012)



- Time to solve the problem is much shorter with the learned controllers (right) than using designed controllers

## Applications: Learning from demonstration

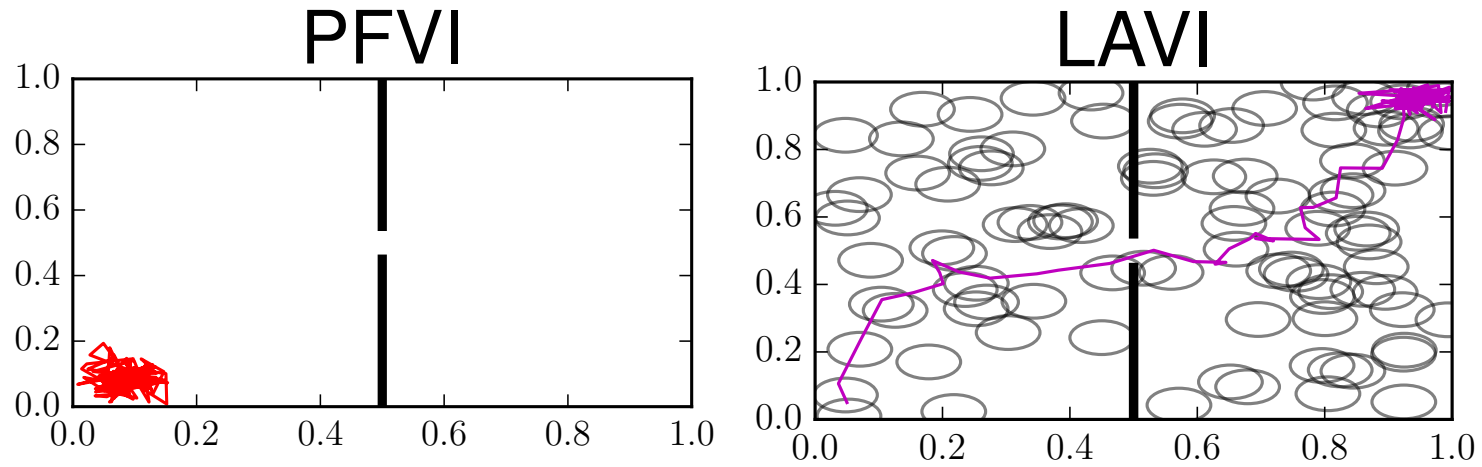
- Very important in human-robot interaction, game playing systems
- Key idea: take human-generated trajectories and decompose them into options
  - Each option policy and model is assumed to rely on a small set of state variables
  - Change-point detection is used to detect option boundaries
  - Once boundaries are detected, feature selection is used to decide which features are important
- Successfully applied to robot block manipulation and to PacMan

Cf. Shim & Thomaz, 2011; Subramanian, Isbell & Thomaz, 2011

## Can we do something simple?

- *Generate a lot of options, then worry about which are useful!*
- More precisely, suppose we have a large set of *landmarks*, i.e. states in the environment, perhaps chosen at random (Mann et al, in press)
- Suppose we have a rough planner which can get to a landmark from its vicinity, by solving a *deterministic relaxation* of the MDP
- We use the landmarks to generate options, then use these in approximate value iteration

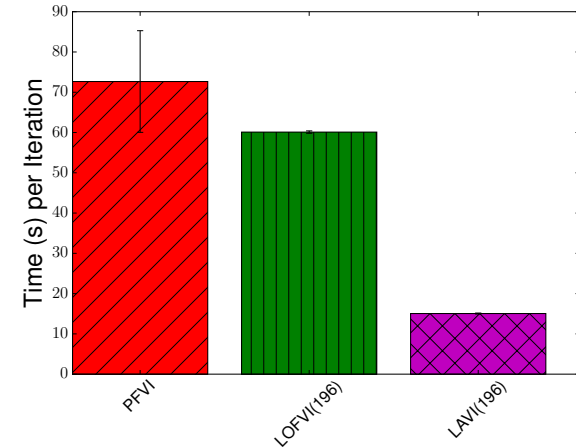
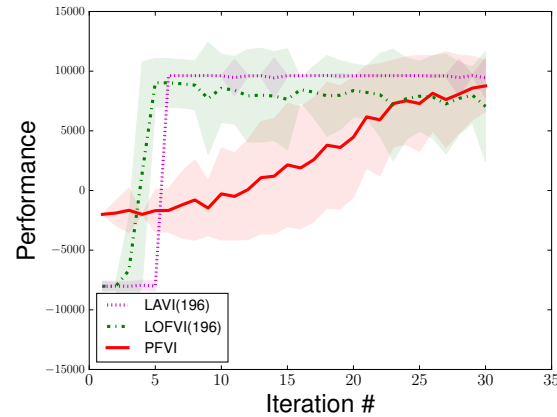
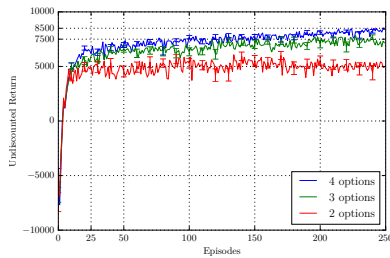
## Illustration of random landmarks



*Landmark-based approximate value iteration gets a good solution much faster!*



# Pinball domain

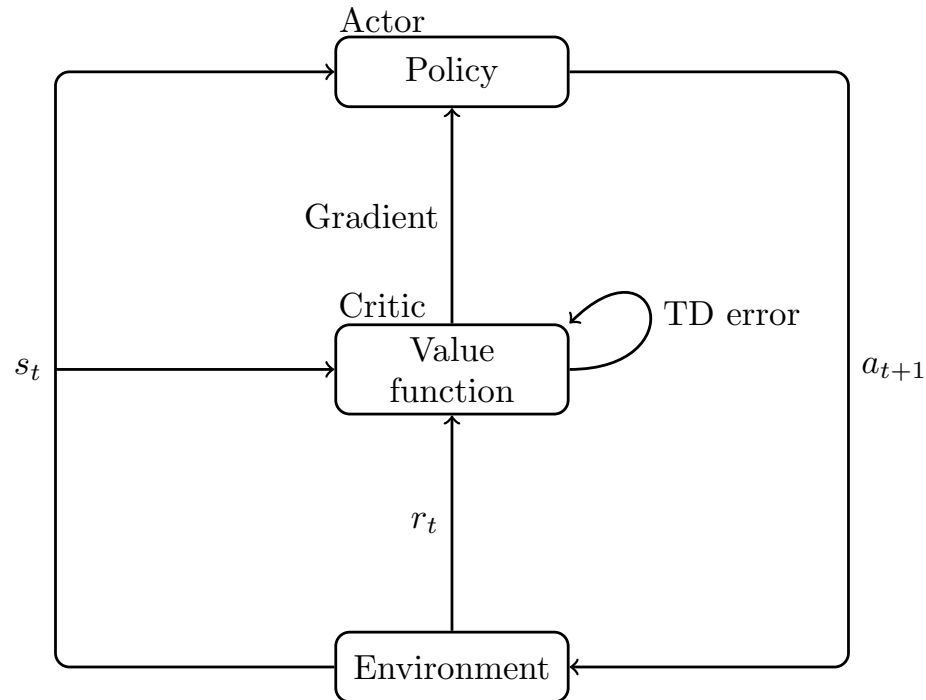


- Random landmarks are very useful
- Representing the value function only at landmark states provides significant speedup
- Consistent with theoretical analysis (in terms of concentrability coefficients)

## Goals of our current work

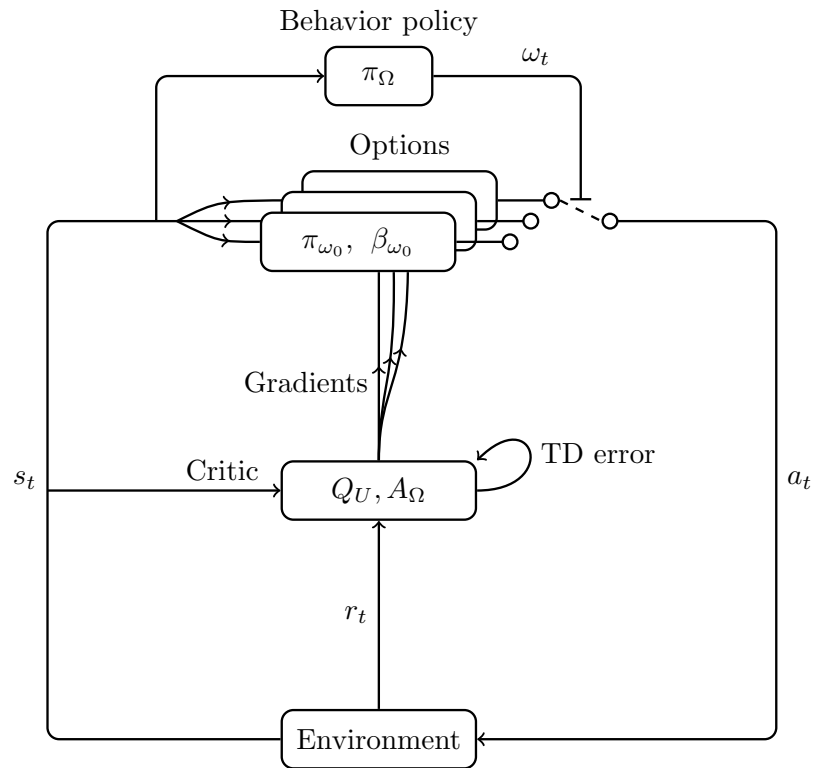
- Explicitly state an *optimization objective* and then solve it to find a set of options
- Handle both *discrete and continuous* set of state and actions
- Learning options should be *continual* (avoid combinatorially-flavored computations)
- Options should provide *improvement within one task* (or at least not cause slow-down...)

# Actor-critic architecture



- Clear optimization objective: average or discounted return
- Continual learning
- Handles both discrete and continuous states and actions

# Option-critic architecture



- Parameterize internal policies and termination conditions
- Policy over options is computed by a separate process (planning, RL, ...)

## Formulation

- The option-value function of a policy over options  $\pi_\Omega$  is given by

$$Q_{\pi_\Omega}(s, \omega) = \sum_a \pi_\omega(a|s) Q_U(s, \omega, a)$$

where

$$Q_U(s, \omega, a) = r_a(s) + \gamma \sum_{s'} P_a(s'|s) U(\omega, s')$$

- The last quantity is the utility from  $s'$  onwards, *given that we arrive in  $s'$  using  $\omega$*

$$U(\omega, s') = (1 - \beta_\omega(s')) Q_{\pi_\Omega}(s', \omega) + \beta_\omega(s') V_{\pi_\Omega}(s')$$

- We parameterize the internal policies by  $\theta$ , as  $\pi_{\omega, \theta}$ , and the termination conditions by  $\nu$ , as  $\beta_{\omega, \nu}$
- *Note that  $\theta$  and  $\nu$  can be shared over the options!*

## Main result: Gradient updates

- Suppose we want to optimize the expected return:  $\mathbb{E} \{ Q_{\pi_{\Omega}}(s, \omega) \}$
- The *gradient wrt the internal policy parameters*  $\theta$  is given by:

$$\mathbb{E} \left\{ \frac{\partial \log \pi_{\omega, \theta}(a|s)}{\partial \theta} Q_U(s, \omega, a) \right\}$$

This has the usual interpretation: *take better primitives more often* inside the option

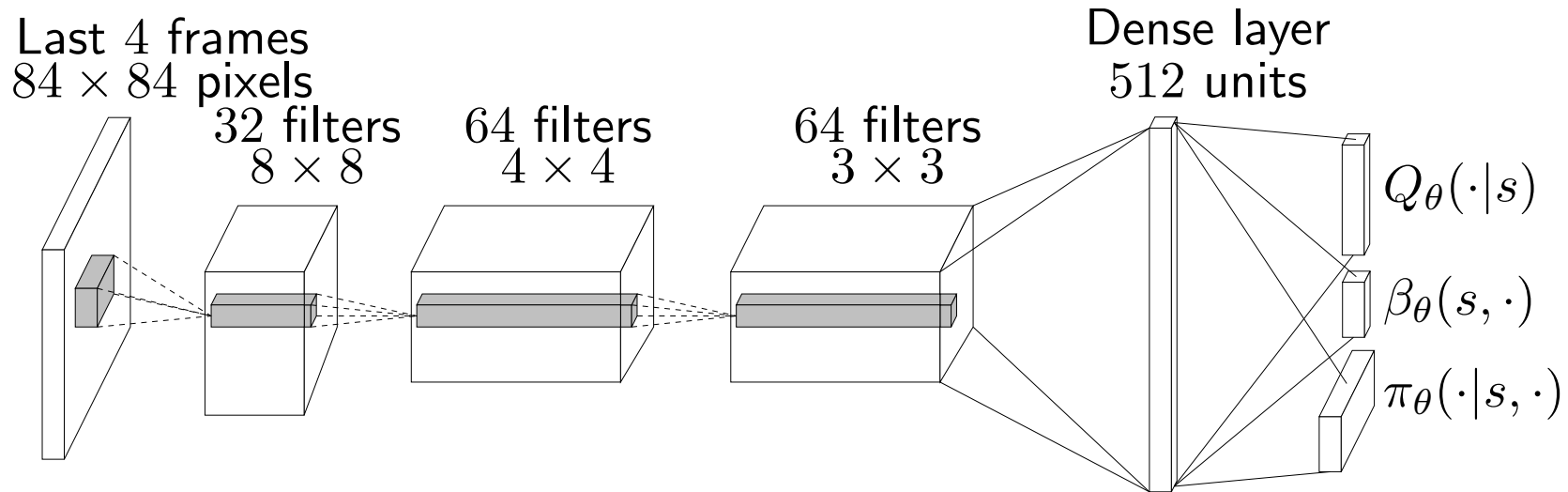
- The *gradient wrt the termination parameters*  $\nu$  is given by:

$$\mathbb{E} \left\{ -\frac{\partial \beta_{\omega, \nu}(s')}{\partial \nu} A_{\pi_{\Omega}}(s', \omega) \right\}$$

where  $A_{\pi_{\Omega}} = Q_{\pi_{\Omega}} - V_{\pi_{\Omega}}$  is the advantage function

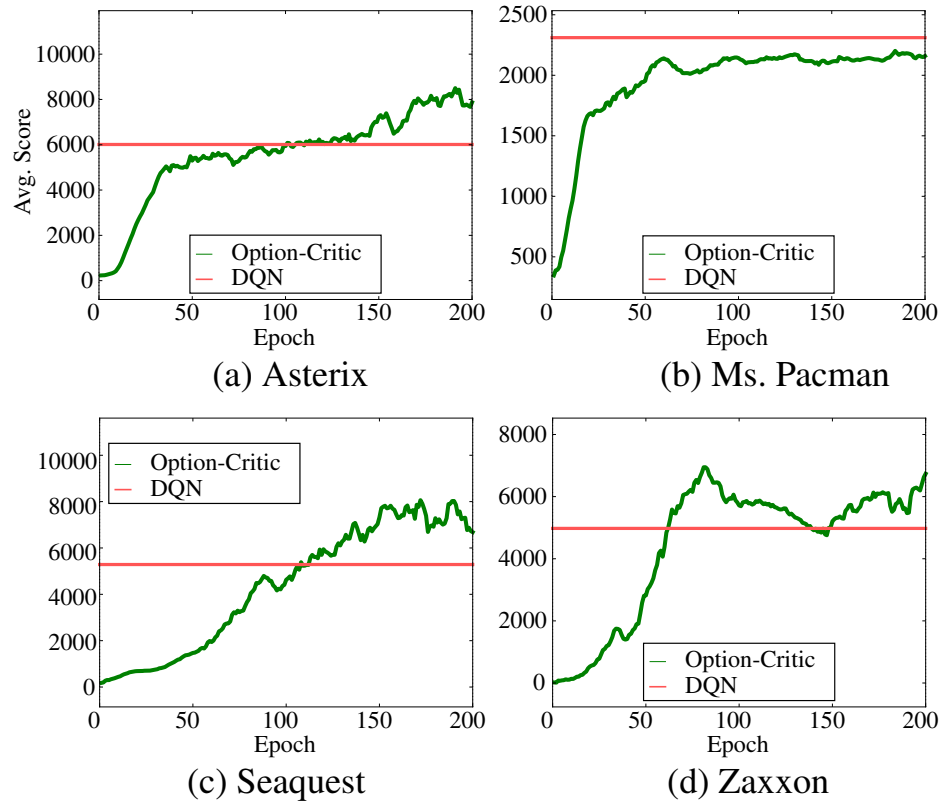
This means that we want to *lengthen options that have a large advantage*

## Option-Critic with Deep Neural Networks



- Atari simulator
- Parameters shared across all options
- Either DQN-style or advantage asynchronous option-critic (A2OC)
- Only number of options needs to be pre-specified

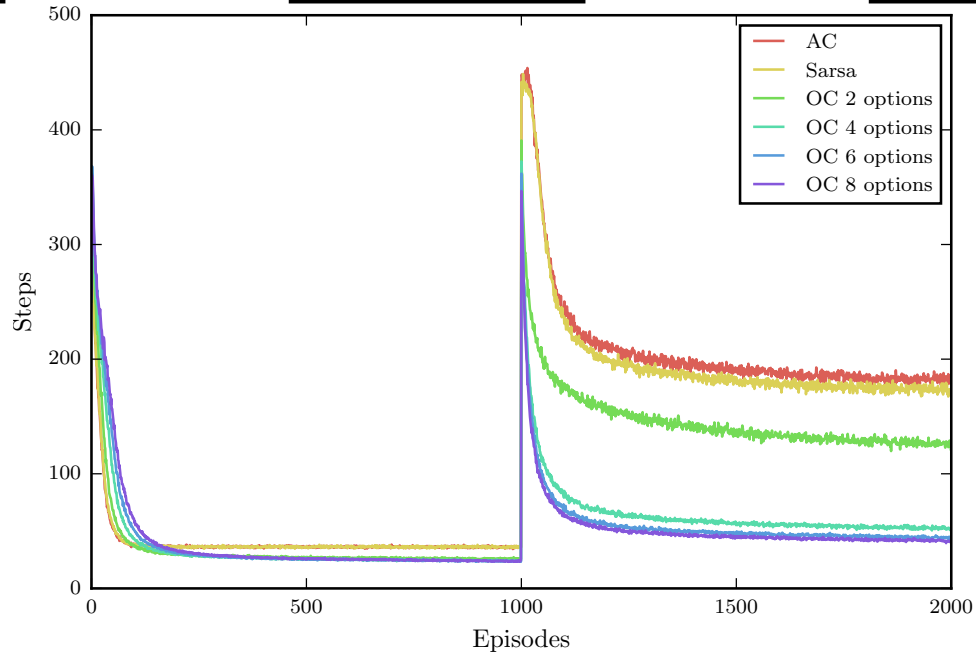
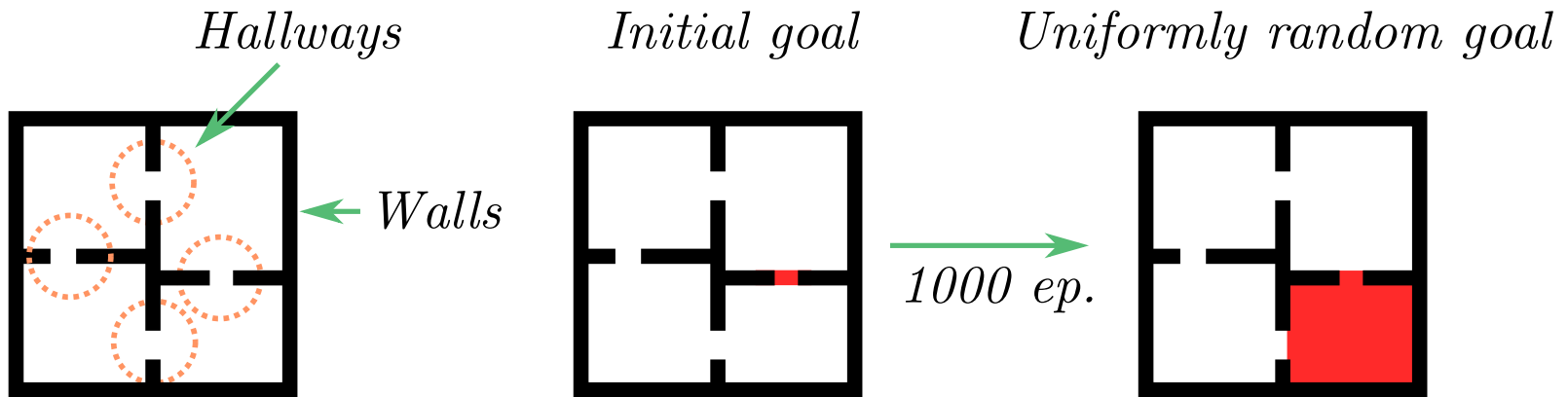
## Results: Atari



- Performance matching or better than DQN *learning within a single task*
- Out of 8 games tested, option-critic does better than published results in 7, with A3C version superior to DQN - mainly due to exploration



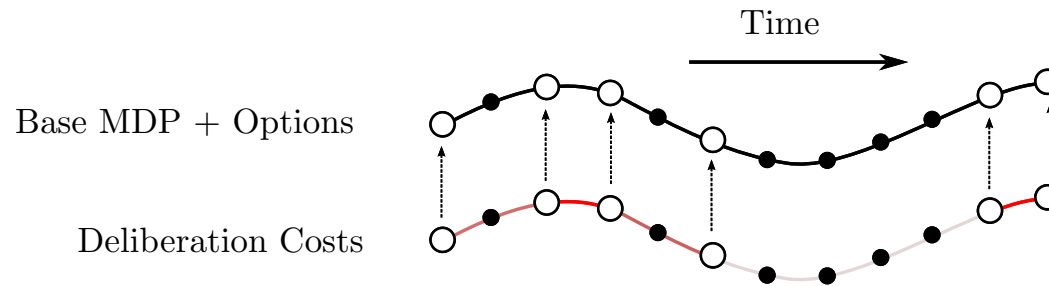
# Results: Transfer in Rooms Domain



# Regularization

- Entropy-based regularization is useful to encourage diversity in the options
- *Length collapse*: options “dissolve” into primitive action over time
- Assumption: *executing a policy is cheap, deciding what to do is expensive*
  - Many choices may need to be evaluated (branching factor over actions)
  - In planning, many next states may need to be considered (branching factor over states)
  - Evaluating the function approximator might be expensive (e.g. if it is a deep net)
- Deliberation is also expensive in animals:
  - Energy consumption (to engage higher-level brain function)
  - Missed opportunity cost: thinking too long means action is delayed

# Deliberation/Switching Cost



- Let  $c(s, \omega) < 0$  be the immediate cost of switching to  $\omega$  in  $s$
- *New value function expresses total deliberation cost:*

$$Q_c(s, \omega) = c(s, \omega) + \sum_{s'} P(s'|s, \omega) \sum_{\omega'} \mu(\omega'|s') Q_c(s', \omega')$$

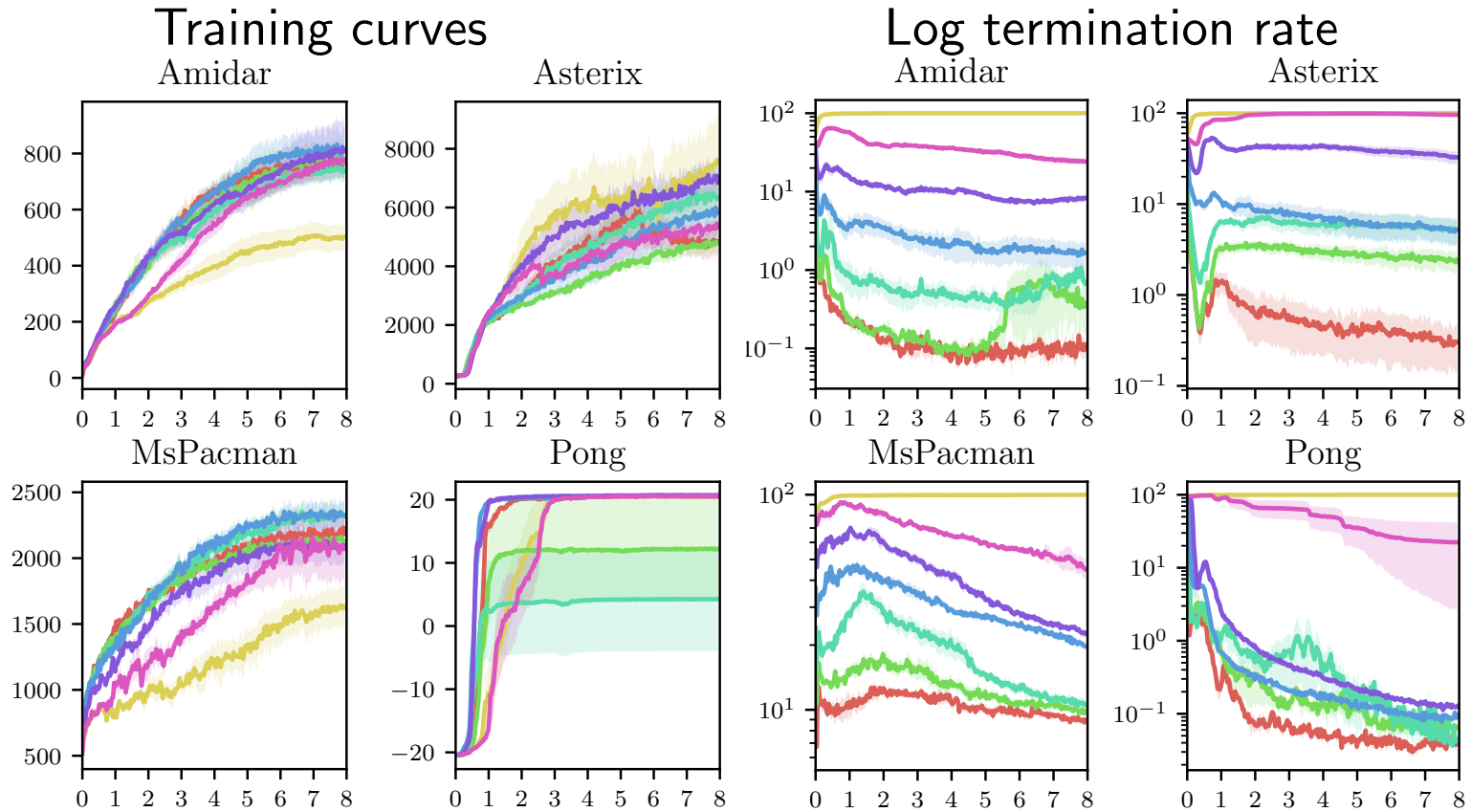
- *New objective: maximize reward with reasonable effort*

$$\max_{\Omega} \mathbb{E} [Q_{\Omega}(s, \omega) + \tau Q_c(s, \omega)]$$

$\tau \geq 0$  controls the trade-off between value and computation effort

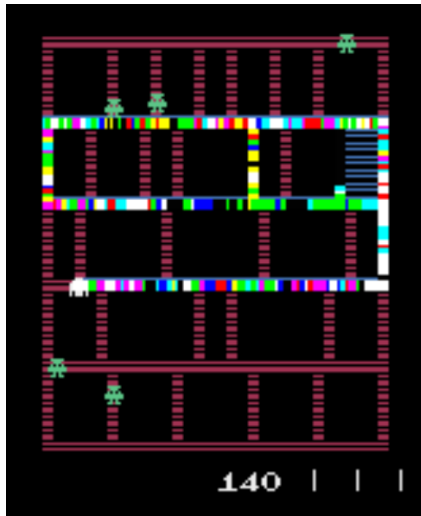
- Can be shown equivalent to requiring that *advantage exceeds a threshold* before switching

# Effect of Deliberation Cost Regularization

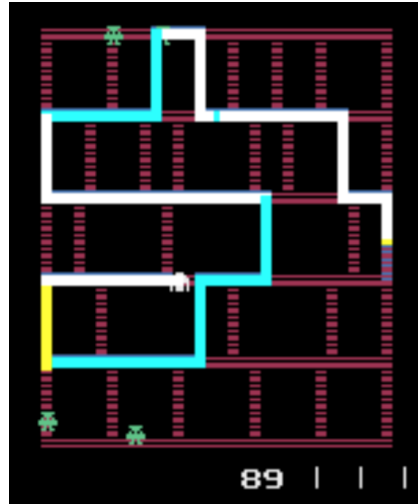


Yellow: no regularization; red: most regularization

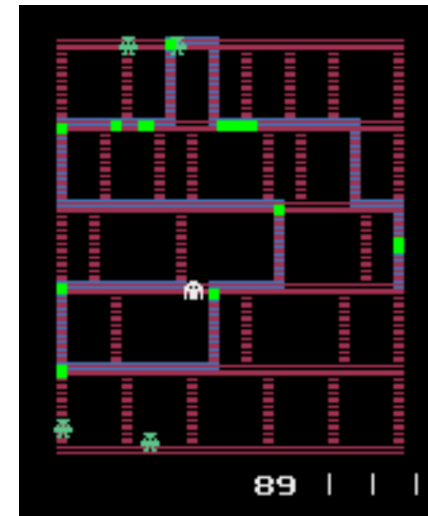
## Example: Amidar



(a) Without a deliberation cost, options terminate instantly and are used in any scenario without specialization.



(b) Options are used for extended periods and in specific scenarios through a trajectory, when using a deliberation cost.



(c) Termination is sparse when using the deliberation cost. The agent terminates options at intersections requiring high level decisions.

- Deliberation costs prevent options from becoming too short
- Terminations are intuitive

# Main Ideas

- *Many big data decision-making applications are a natural fit for reinforcement learning algorithms*
- Key problems in reinforcement learning are the large number of states, the large number of actions, and the fast time scale for decisions
- *Temporal abstraction* helps by controlling the branching factor, decreasing the number of decisions, and reducing the number of states at which decision need to be made
- As a result the speed of learning and planning increase considerably
- *Options and models may not need to be very carefully crafted* (just use “big data” approach)