# Computationally Efficient Relational Reinforcement Learning

Mitchell Keith Bloch

University of Michigan
2260 Hayward Street
Ann Arbor, MI. 48109-2121
bazald@umich.edu

May 16-17, 2018

# The Problem from 10,000 Feet

**Research Interest**

I seek to develop agents that can learn from a reward signal

# The Problem from 10,000 Feet

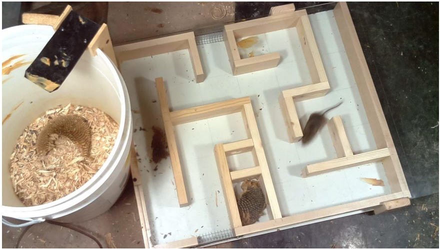
**Research Interest**

I seek to develop agents that can learn from a reward signal

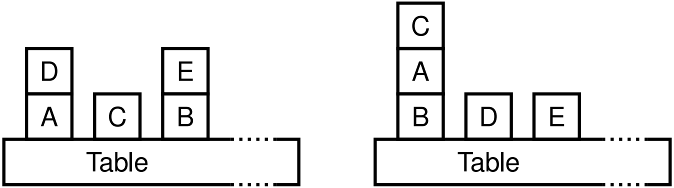
# The Problem from 10,000 Feet

**Research Interest**

I seek to develop agents that can learn from a reward signal



# Blocks World – Objective: **`Exact`**



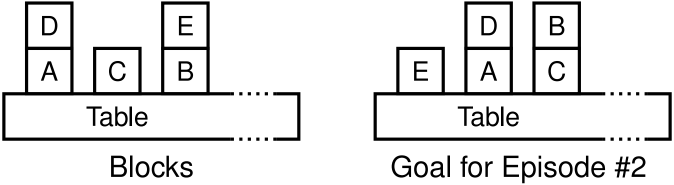

Blocks

Goal for Episode #1

Complete state description visual

1. Full representation of the goal presented by the environment
2. Variable goals and potentially numbers of blocks each episode
3. Relatively complex training goal vs
   - **`Stack`** – Creating a tower out of all the blocks
   - **`Unstack`** – Placing all blocks on the table
   - **`On(a,b)`** – Placing one specific block on top of another
4. 8 features plus 22 distractor features

# Blocks World – Objective: **`Exact`**




Complete state description visual

1. Full representation of the goal presented by the environment
2. Variable goals and potentially numbers of blocks each episode
3. Relatively complex training goal vs
   - **`Stack`** – Creating a tower out of all the blocks
   - **`Unstack`** – Placing all blocks on the table
   - **`On(a,b)`** – Placing one specific block on top of another
4. 8 features plus 22 distractor features

# Temporal Difference Learning

### Learning Mechanism

Temporal Difference (TD) methods for Reinforcement Learning (RL) are generally applicable and can support online learning

# Temporal Difference Learning

**Learning Mechanism**

Temporal Difference (TD) methods for Reinforcement Learning (RL) are generally applicable and can support online learning

**Sarsa (On Policy)**

$$Q(s,a) \overset{\alpha}{\leftarrow} r + \gamma Q(s',a')$$

**Q-learning (Off policy)**

$$Q(s,a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')$$

# Temporal Difference Learning

## Learning Mechanism

Temporal Difference (TD) methods for Reinforcement Learning (RL) are generally applicable and can support online learning

## Sarsa (On Policy)

$$Q(s,a) \xleftarrow{\alpha} r + \gamma Q(s',a')$$

## Q-learning (Off policy)

$$Q(s,a) \xleftarrow{\alpha} r + \gamma \max_{a' \in \mathcal{A}} Q(s',a')$$

## Greedy-GQ($\lambda$)

[Maei and Sutton, 2010]

# Q-functions

**TD methods over ...**

What is $Q(s, a)$?

# Q-functions

**TD methods over …**

What is $Q(s, a)$?

**Tabular RL**

$Q(s, a)$ can map each state-action pair to a unique value called a *Q-value*

# Q-functions

**TD methods over ...**

What is $Q(s, a)$?

**Tabular RL**

$Q(s, a)$ can map each state-action pair to a unique value called a *Q-value*

**Linear Function Approximation**

$Q(s, a)$ can map each state-action pair to a sum of *weights* that are shared between different state-action pairs

$$Q(s, a) = \sum_{i=1}^{n} \phi_i(s, a) w_i$$

# Features?

## Linear Function Approximation

Where do features, $\phi_i(s, a)$, come from?

# Features?

**Linear Function Approximation**

Where do features, $\phi_i(s, a)$, come from?

**TD Methods**

TD methods answer the problem of how to learn over features, but do nothing to answer the problem of where features should come from

# Features?

**Linear Function Approximation**

Where do features, $\phi_i(s, a)$, come from?

**TD Methods**

TD methods answer the problem of how to learn over features, but do nothing to answer the problem of where features should come from

**Tile Codings**

One answer to this problem is to use tile codings to partition the state-action space

# Tile Coding

## One 8x8 tiling

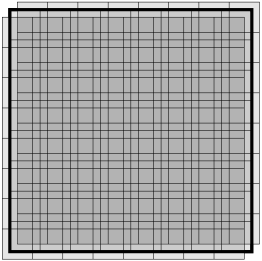
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1.0 | -1.1 | -1.2 | -1.0 | 2.1 | 2.0 | 3.1 | 5.1 |
| -0.7 | -1.2 | -1.0 | -1.1 | 1.8 | 2.0 | 2.9 | 4.1 |
| -2.9 | -2.8 | -1.0 | -1.1 | 2.0 | 1.9 | 2.9 | 3.2 |
| -4.9 | -3.1 | -1.2 | -0.9 | 2.1 | 2.1 | 2.1 | 2.0 |
| -3.2 | -2.8 | 0.2 | -0.1 | 0.9 | 1.0 | 1.1 | 1.2 |
| -2.1 | -1.8 | 0.0 | 0.1 | 1.0 | 0.9 | 1.1 | 1.0 |
| -1.1 | -0.9 | 0.0 | -0.2 | 0.9 | 1.0 | 1.1 | 0.8 |
| -1.2 | -1.0 | 0.2 | 0.1 | 1.1 | 1.0 | 0.9 | 1.0 |

# Tile Coding

## One 8x8 tiling

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1.0 | -1.1 | -1.2 | -1.0 | 2.1 | 2.0 | 3.1 | 5.1 |
| -0.7 | -1.2 | -1.0 | -1.1 | 1.8 | 2.0 | 2.9 | 4.1 |
| -2.9 | -2.8 | -1.0 | -1.1 | 2.0 | 1.9 | 2.9 | 3.2 |
| -4.9 | -3.1 | -1.2 | -0.9 | 2.1 | 2.1 | 2.1 | 2.0 |
| -3.2 | -2.8 | 0.2 | -0.1 | 0.9 | 1.0 | 1.1 | 1.2 |
| -2.1 | -1.8 | 0.0 | 0.1 | 1.0 | 0.9 | 1.1 | 1.0 |
| -1.1 | -0.9 | 0.0 | -0.2 | 0.9 | 1.0 | 1.1 | 0.8 |
| -1.2 | -1.0 | 0.2 | 0.1 | 1.1 | 1.0 | 0.9 | 1.0 |

## Three 8x8 tilings

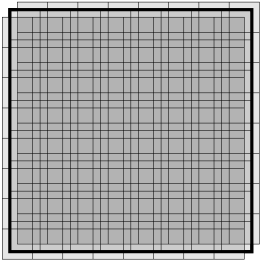


# Tile Coding

One 8x8 tiling

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1.0 | -1.1 | -1.2 | -1.0 | 2.1 | 2.0 | 3.1 | 5.1 |
| -0.7 | -1.2 | -1.0 | -1.1 | 1.8 | 2.0 | 2.9 | 4.1 |
| -2.9 | -2.8 | -1.0 | -1.1 | 2.0 | 1.9 | 2.9 | 3.2 |
| -4.9 | -3.1 | -1.2 | -0.9 | 2.1 | 2.1 | 2.1 | 2.0 |
| -3.2 | -2.8 | 0.2 | -0.1 | 0.9 | 1.0 | 1.1 | 1.2 |
| -2.1 | -1.8 | 0.0 | 0.1 | 1.0 | 0.9 | 1.1 | 1.0 |
| -1.1 | -0.9 | 0.0 | -0.2 | 0.9 | 1.0 | 1.1 | 0.8 |
| -1.2 | -1.0 | 0.2 | 0.1 | 1.1 | 1.0 | 0.9 | 1.0 |

Three 8x8 tilings



Adaptive Tile Coding (ATC)

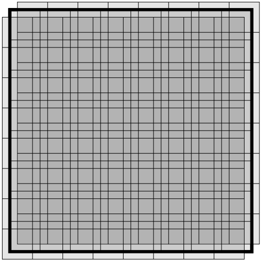
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1.0 | -1.1 | -1.2 | -1.0 | 2.1 | 2.0 | 3.1 | 5.1 |
| -0.7 | -1.2 | -1.0 | -1.1 | 1.8 | 2.0 | 2.9 | 4.1 |
| -2.9 | -2.8 | -1.0 | -1.1 | 2.0 | 1.9 | 2.9 | 3.2 |
| -4.9 | -3.1 | -1.2 | -0.9 | 2.1 | 2.1 | 2.1 | 2.0 |
| -3.2 | -2.8 | 0.2 | -0.1 | 0.9 | 1.0 | 1.1 | 1.2 |
| -2.1 | -1.8 | 0.0 | 0.1 | 1.0 | 0.9 | 1.1 | 1.0 |
| -1.1 | -0.9 | 0.0 | -0.2 | 0.9 | 1.0 | 1.1 | 0.8 |
| -1.2 | -1.0 | 0.2 | 0.1 | 1.1 | 1.0 | 0.9 | 1.0 |

# Tile Coding

One 8x8 tiling

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1.0 | -1.1 | -1.2 | -1.0 | 2.1 | 2.0 | 3.1 | 5.1 |
| -0.7 | -1.2 | -1.0 | -1.1 | 1.8 | 2.0 | 2.9 | 4.1 |
| -2.9 | -2.8 | -1.0 | -1.1 | 2.0 | 1.9 | 2.9 | 3.2 |
| -4.9 | -3.1 | -1.2 | -0.9 | 2.1 | 2.1 | 2.1 | 2.0 |
| -3.2 | -2.8 | 0.2 | -0.1 | 0.9 | 1.0 | 1.1 | 1.2 |
| -2.1 | -1.8 | 0.0 | 0.1 | 1.0 | 0.9 | 1.1 | 1.0 |
| -1.1 | -0.9 | 0.0 | -0.2 | 0.9 | 1.0 | 1.1 | 0.8 |
| -1.2 | -1.0 | 0.2 | 0.1 | 1.1 | 1.0 | 0.9 | 1.0 |

Three 8x8 tilings



Adaptive Hierarchical Tile Coding (aHTC)

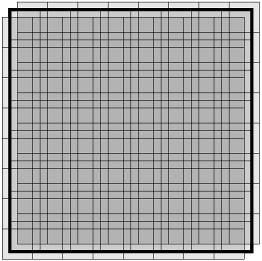
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1.0 | -1.1 | -1.2 | -1.0 | 2.1 | 2.0 | 3.1 | 5.1 |
| -0.7 | -1.2 | -1.0 | -1.1 | 1.8 | 2.0 | 2.9 | 4.1 |
| -2.9 | -2.8 | -1.0 | -1.1 | 2.0 | 1.9 | 2.9 | 3.2 |
| -4.9 | -3.1 | -1.2 | -0.9 | 2.1 | 2.1 | 2.1 | 2.0 |
| -3.2 | -2.8 | 0.2 | -0.1 | 0.9 | 1.0 | 1.1 | 1.2 |
| -2.1 | -1.8 | 0.0 | 0.1 | 1.0 | 0.9 | 1.1 | 1.0 |
| -1.1 | -0.9 | 0.0 | -0.2 | 0.9 | 1.0 | 1.1 | 0.8 |
| -1.2 | -1.0 | 0.2 | 0.1 | 1.1 | 1.0 | 0.9 | 1.0 |

# Tile Coding

One 8x8 tiling

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1.0 | -1.1 | -1.2 | -1.0 | 2.1 | 2.0 | 3.1 | 5.1 |
| -0.7 | -1.2 | -1.0 | -1.1 | 1.8 | 2.0 | 2.9 | 4.1 |
| -2.9 | -2.8 | -1.0 | -1.1 | 2.0 | 1.9 | 2.9 | 3.2 |
| -4.9 | -3.1 | -1.2 | -0.9 | 2.1 | 2.1 | 2.1 | 2.0 |
| -3.2 | -2.8 | 0.2 | -0.1 | 0.9 | 1.0 | 1.1 | 1.2 |
| -2.1 | -1.8 | 0.0 | 0.1 | 1.0 | 0.9 | 1.1 | 1.0 |
| -1.1 | -0.9 | 0.0 | -0.2 | 0.9 | 1.0 | 1.1 | 0.8 |
| -1.2 | -1.0 | 0.2 | 0.1 | 1.1 | 1.0 | 0.9 | 1.0 |

Three 8x8 tilings



Adaptive Hierarchical Tile Coding (aHTC)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -1.0 | -1.1 | -1.2 | -1.0 | 2.1 | 2.0 | 3.1 | 5.1 |
| -0.7 | -1.2 | -1.0 | -1.1 | 1.8 | 2.0 | 2.9 | 4.1 |
| -2.9 | -2.8 | -1.0 | -1.1 | 2.0 | 1.9 | 2.9 | 3.2 |
| -4.9 | -3.1 | -1.2 | -0.9 | 2.1 | 2.1 | 2.1 | 2.0 |
| -3.2 | -2.8 | 0.2 | -0.1 | 0.9 | 1.0 | 1.1 | 1.2 |
| -2.1 | -1.8 | 0.0 | 0.1 | 1.0 | 0.9 | 1.1 | 1.0 |
| -1.1 | -0.9 | 0.0 | -0.2 | 0.9 | 1.0 | 1.1 | 0.8 |
| -1.2 | -1.0 | 0.2 | 0.1 | 1.1 | 1.0 | 0.9 | 1.0 |

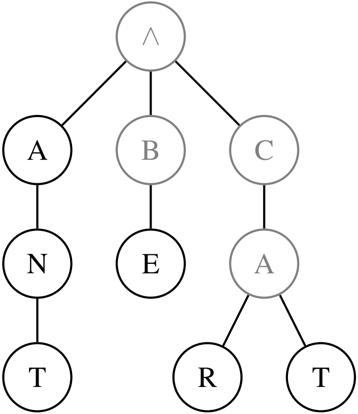
## Learning Efficiency

The offers generalization for states that share a tile

# Implementation: $k$-Dimensional Tries ($k$-d Tries)

**Typical Use**

Efficient English dictionary storage and lookup



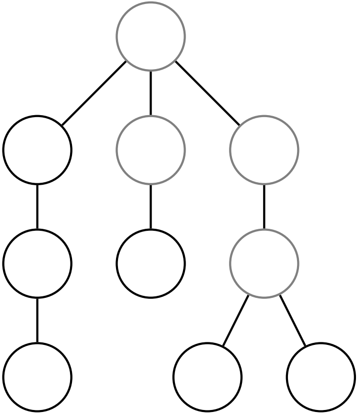
# Implementation: $k$-Dimensional Tries ($k$-d Tries)

**Typical Use**

Efficient English dictionary storage and lookup

**My Use**

Can also be used for efficient representation of an ATC or HTC



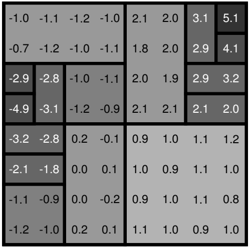
# Implementation: *k*-Dimensional Tries (*k*-d Tries)

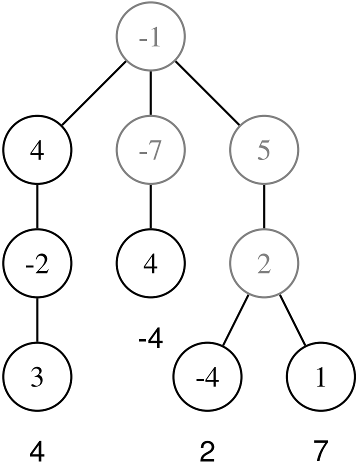
**Typical Use**

Efficient English dictionary storage and lookup

**My Use**

Can also be used for efficient representation of an ATC or HTC



# Adaptive Tile Coding

## ATC/HTC Trie Mapping

- Less refined tilings correspond to conjunctions of few features
- More refined tilings correspond to conjunctions of many features
- The most refined tilings correspond to *fringe* nodes
  i.e. candidate conjunctions for inclusion in the value function

# Adaptive Tile Coding

## ATC/HTC Trie Mapping

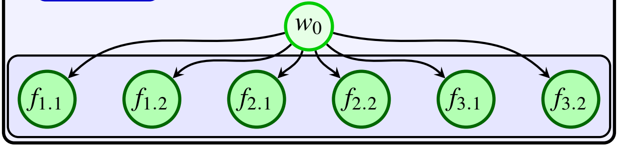
- Less refined tilings correspond to conjunctions of few features
- More refined tilings correspond to conjunctions of many features
- The most refined tilings correspond to *fringe* nodes
  i.e. candidate conjunctions for inclusion in the value function

**aHTC View**

$w_0$

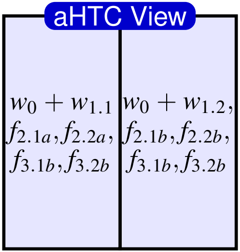
$f_{1.1}, f_{1.2}, f_{2.1},$
$f_{2.2}, f_{3.1}, f_{3.2}$

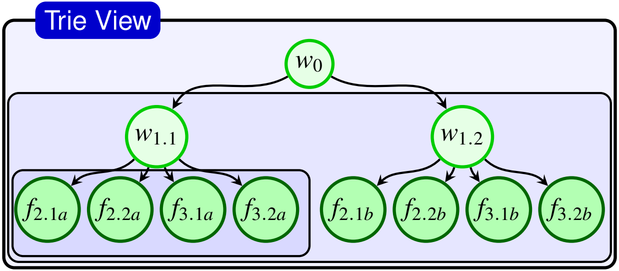
**Trie View**



## ATC/HTC Trie Mapping

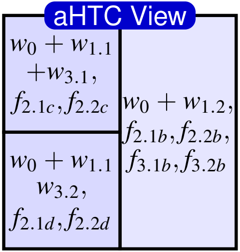
- Less refined tilings correspond to conjunctions of few features
- More refined tilings correspond to conjunctions of many features
- The most refined tilings correspond to *fringe* nodes
  i.e. candidate conjunctions for inclusion in the value function

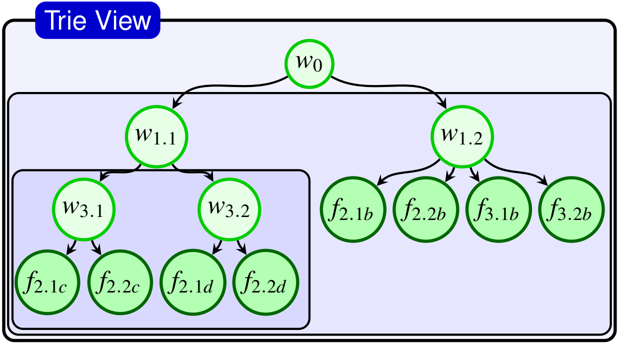


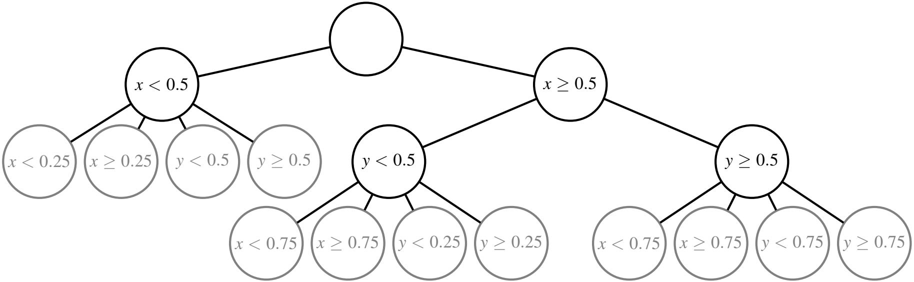
## ATC/HTC Trie Mapping

- Less refined tilings correspond to conjunctions of few features
- More refined tilings correspond to conjunctions of many features
- The most refined tilings correspond to *fringe* nodes
  i.e. candidate conjunctions for inclusion in the value function



# Adaptive *k*-d Trie Representation



# Relational Representations

**Concept**

First Order Logical Decison Tree (FOLDT)

**Concept**

First Order Logical Decison Tree (FOLDT)

**Feature 2**

on($b$, $a$)

# Relational Representations

**Concept**

First Order Logical Decison Tree (FOLDT)

**Feature 2**

on($b$, $a$)

**Feature 5**

clear($c$)

# Relational Representations

**Concept**

First Order Logical Decison Tree (FOLDT)

**Feature 2**

on($b$, $a$)

**Feature 5**

clear($c$)

**Feature 8**

$\neg$clear($a$)

# Relational Representations

**Concept**

First Order Logical Decison Tree (FOLDT)

**Problem**

*k*-d Tries do not effectively support FOLDT implementation due to variable binding problem!

**Feature 2**

on($b$, $a$)

**Feature 5**

clear($c$)

**Feature 8**

$\neg$clear($a$)

# Relational Representations

**Concept**

First Order Logical Decison Tree (FOLDT)

**Problem**

*k*-d Tries do not effectively support FOLDT implementation due to variable binding problem!

**Solution**

I observed that a FOLDT could be embedded in a Rete for efficient RRL implementation

**Feature 2**

on($b$, $a$)

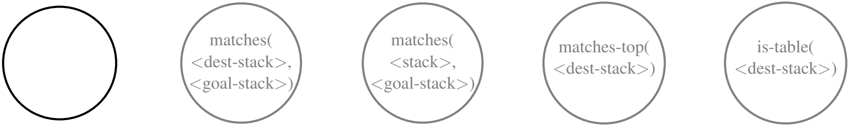
**Feature 5**

clear($c$)

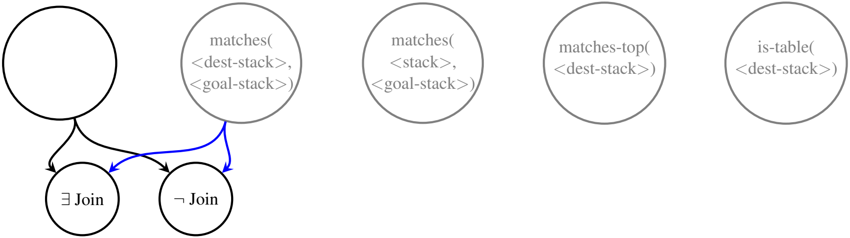
**Feature 8**

¬clear($a$)

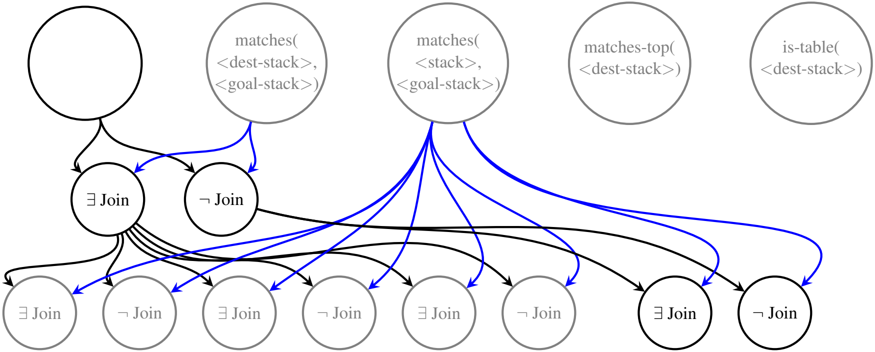
# Adaptive Rete Representation



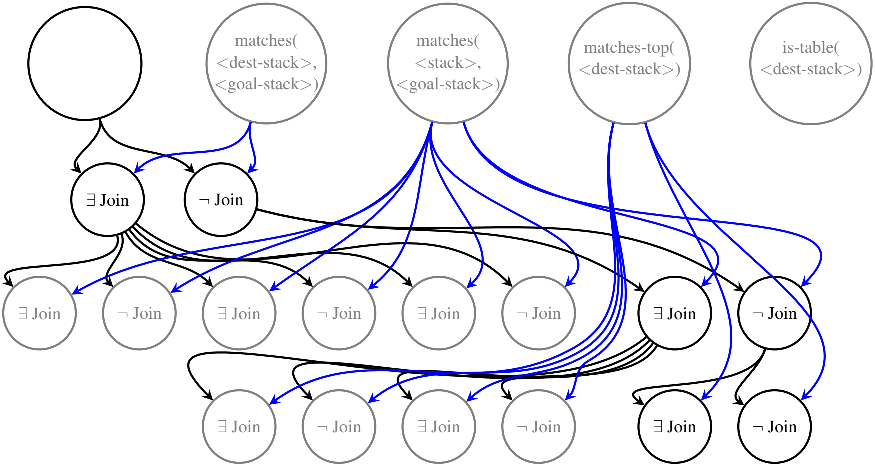
# Adaptive Rete Representation

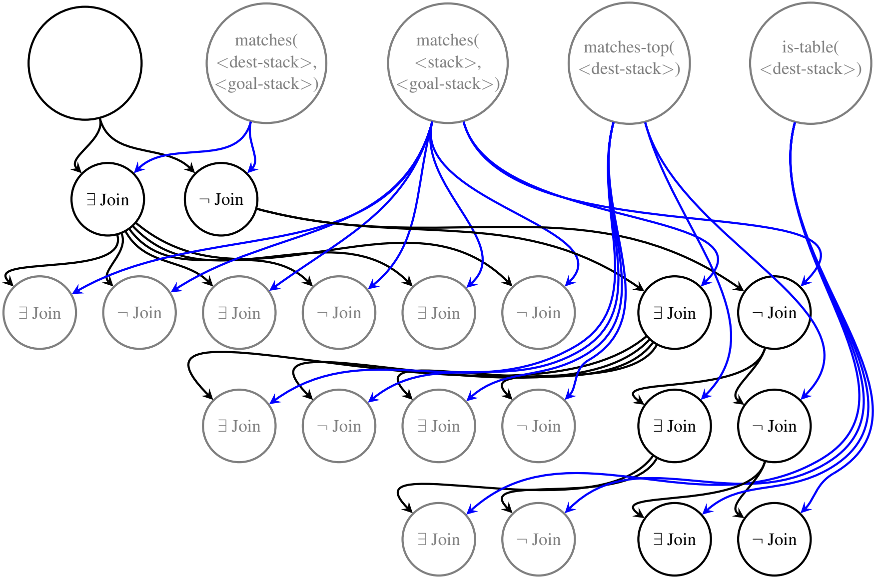


# Adaptive Rete Representation



# Adaptive Rete Representation




matches(
<dest-stack>,
<goal-stack>)
matches(
<stack>,
<goal-stack>)
matches-top(
<dest-stack>)
is-table(
<dest-stack>)
∃ Join
¬ Join
∃ Join
¬ Join
∃ Join
¬ Join
∃ Join
¬ Join
∃ Join
¬ Join
∃ Join
¬ Join
∃ Join
¬ Join
∃ Join
¬ Join
∃ Join
¬ Join
∃ Join
¬ Join

# Exploration of Refinement Criteria

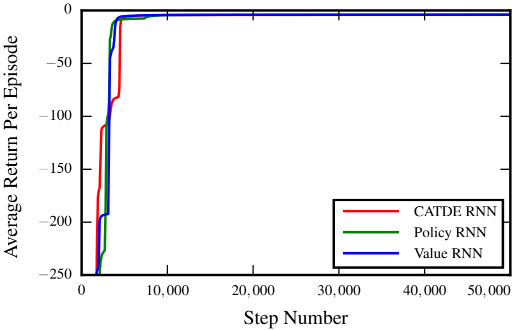
## Criteria

1. Cumulative Absolute Temporal Difference Error (CATDE) – Maximal error accumulation
   - Focus on regions of high activity and error.
   - Track TD error experienced at each leaf in the value function.
   - The nodes with highest error are eligible for refinement when their features match.
2. Policy – Maximal change in $\pi(s, a)$
   - Focus on modifying policy (Whiteson 2007)
   - Choose features which maximize the change in the greedy set of actions.
3. Value – Maximal change in $Q(s, a)$
   - Focus on improving value estimates (Whiteson 2007)
   - Choose features which maximize value spread on refinement.

# Exact with Refinement Only – No Distractors

- “Refinement only” gives a baseline
- All criteria okay with no distractors

## Learning



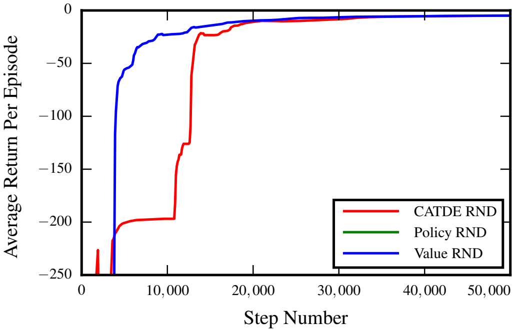

## Average Return Per Episode & Wall Clock Time Per Step

| Criterion at $50,000$ | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –4.06 | 1.38ms | 23.0 |
| Policy Criterion | –3.84 | 0.91ms | 24.7 |
| Value Criterion | –3.97 | 1.33ms | 25.6 |

# `Exact` with Refinement Only – With Distractors

- Value criterion does best with distractors
- Number of weights skyrockets
- Policy criterion performance too low to appear

## Learning



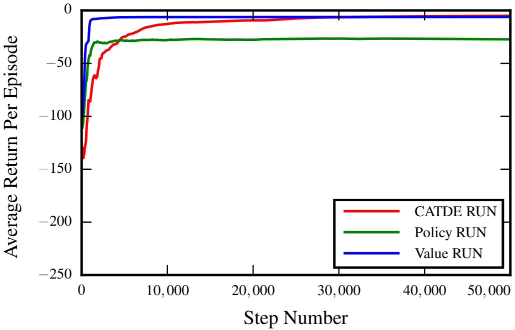

## Average Return Per Episode & Wall Clock Time Per Step

| Criterion at 50,000 | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –4.93 | 18.8ms | 1,487.8 |
| Policy Criterion | –639 | 21.7ms | 1,318.4 |
| Value Criterion | –4.83 | 19.0ms | 1,459.5 |

# `Exact` with Rerefinement – No Distractors

- CATDE does best with unrestricted rerefinement
- Policy does not converge

## Learning



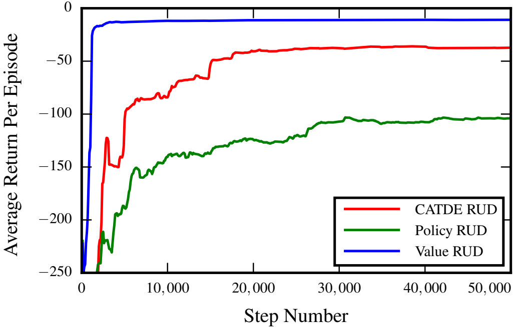

## Average Return Per Episode & Wall Clock Time Per Step

| Criterion at 50,000 | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –5.03 | 1.65ms | 16.3 |
| Policy Criterion | –27.2 | 1.06ms | 4.1 |
| Value Criterion | –6.13 | 1.10ms | 5.94 |

# `Exact` with Rerefinement – With Distractors

- Until you include distractors – then value does best
- Number of weights persistent in the system is low due to thrashing
- Fast execution as a result of few weights

## Learning




## Average Return Per Episode & Wall Clock Time Per Step

| Criterion at $50,000$ | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –37.3 | 3.27ms | 7.58 |
| Policy Criterion | –104 | 1.79ms | 3.15 |
| Value Criterion | –10.9 | 2.20ms | 4.75 |

# `Exact` with Rerefinement

## Functionality So Far

- Demonstrated efficacy in absence of distractors
- Demonstrated computationally efficient refinement and rerefinement

# `Exact` with Rerefinement

## Functionality So Far

- Demonstrated efficacy in absence of distractors
- Demonstrated computationally efficient refinement and rerefinement

## Issues

- Poor quality of learning with distractors
- Incomplete convergence without distractors
- No convergence with distractors

# `Exact` with Rerefinement

## Functionality So Far

- Demonstrated efficacy in absence of distractors
- Demonstrated computationally efficient refinement and rerefinement

## Issues

- Poor quality of learning with distractors
- Incomplete convergence without distractors
- No convergence with distractors

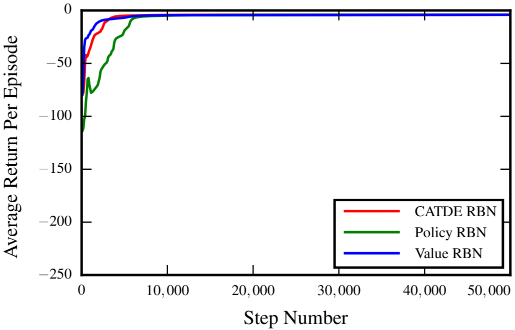
## What's Next

- Demonstrate flexibility of the architecture
- Blacklists
- Boost
- Concrete

# Exact with Rerefinement and Blacklists – No Dist.

- Most obvious strategy to avoid thrashing
- Works fairly well in the absence of distractors

**Learning**



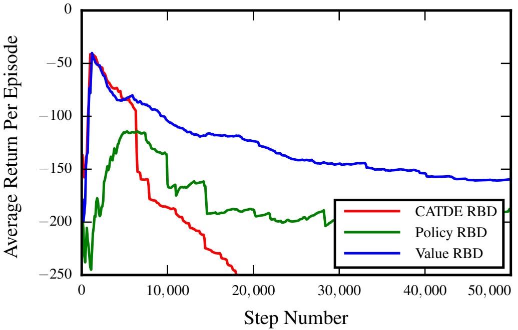

**Average Return Per Episode & Wall Clock Time Per Step**

| Criterion at 50,000 | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –3.94 | 1.51ms | 26.5 |
| Policy Criterion | –4.04 | 1.20ms | 25.9 |
| Value Criterion | –4.13 | 1.41ms | 26.8 |

# **Exact** with Rerefinement and Blacklists – w/ Dist.

- Works poorly with distractors
- Best features likely to be tried and blacklisted first
- Converge on worst value function structure

## Learning



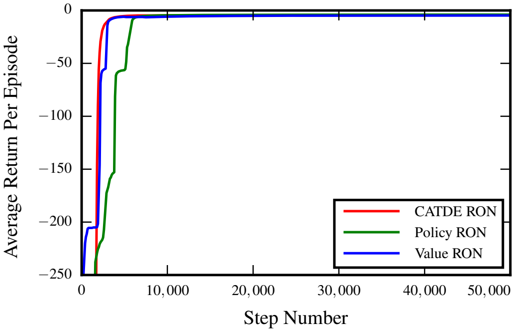

## Average Return Per Episode & Wall Clock Time Per Step

| Criterion at 50,000 | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –407 | 4.72ms | 30.9 |
| Policy Criterion | –187 | 8.55ms | 87.1 |
| Value Criterion | –159 | 5.55ms | 44.7 |

# `Exact` with Rerefinement and Boost – No Dist.

- Gradually increase the likelihood of reselection instead – opposite of blacklists
- Generally a little slower to converge than when using blacklists

## Learning



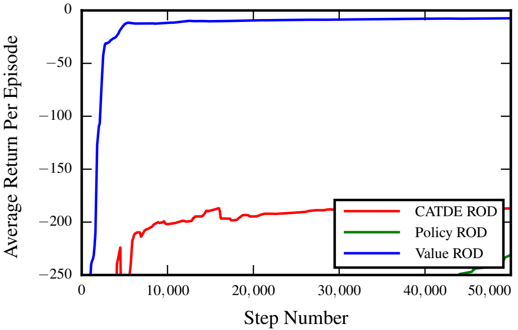

## Average Return Per Episode & Wall Clock Time Per Step

| Criterion at $50,000$ | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –4.14 | 1.77ms | 22.7 |
| Policy Criterion | –3.95 | 2.22ms | 24.8 |
| Value Criterion | –4.78 | 1.56ms | 15.1 |

# `Exact` with Rerefinement and Boost – w/ Dist.

- Much slower with distractors
- However, now the value function comes close to converging

## Learning



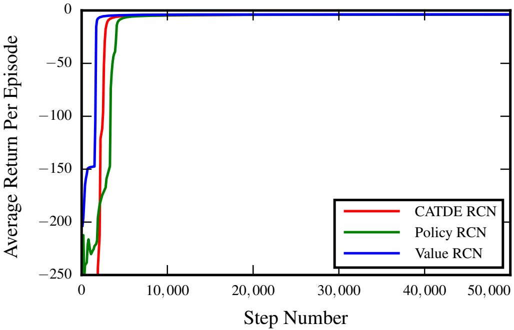

## Average Return Per Episode & Wall Clock Time Per Step

| Criterion at 50,000 | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –187 | 12.4 ms | 73.2 |
| Policy Criterion | –231 | 33.5 ms | 218 |
| Value Criterion | –7.42 | 6.91ms | 26.2 |

# `Exact` with Reref. & Boost & Concrete – No Dist.

- Value criterion gives best performance
- Lowest CPU cost

**Learning**



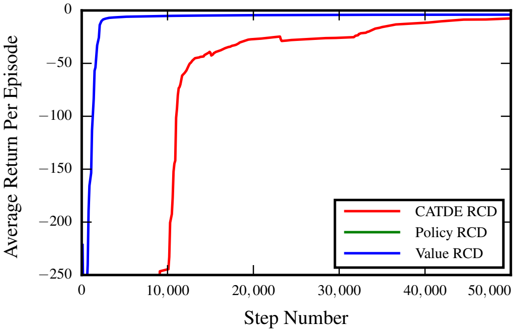

**Average Return Per Episode & Wall Clock Time Per Step**

| Criterion at 50,000 | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –3.99 | 0.605ms | 23.6 |
| Policy Criterion | –3.91 | 0.645ms | 24.0 |
| Value Criterion | –3.71 | 0.612ms | 26.4 |

# `Exact` with Reref. & Boost & Concrete – w/ Dist.

- Value gives best performance even with distractors
- Does so with fewer weights and lower CPU cost

## Learning



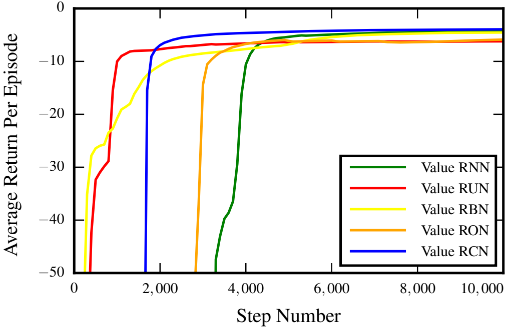

## Average Return Per Episode & Wall Clock Time Per Step

| Criterion at 50,000 | ARtPE | WCTPS | # Weights |
|---|---|---|---|
| CATDE | –7.60 | 19.6ms | 596 |
| Policy Criterion | –1,100 | 10.2ms | 238 |
| Value Criterion | –3.97 | 12.3ms | 284 |

# `Exact` with the Value Criterion – No Dist.

- Unrestricted rerefinement does the best between 1,000 and 3,000 steps
- Rerefinement with boost and concrete overtakes it from 3,000 steps on

**Learning**

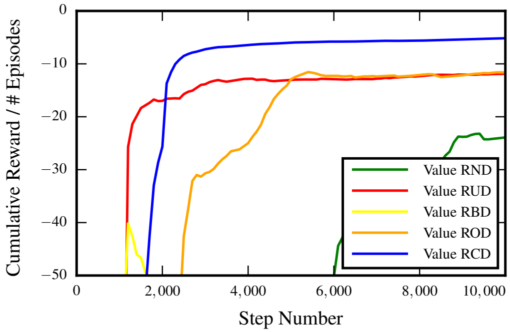



N – No rerefinement
U – Unrestricted rerefinement
B – Blacklists
O – bOost
C – boost with Concrete

# `Exact` with the Value Criterion – w/ Dist.

- Unrestricted rerefinement does the best between 1,000 and 2,000 steps
- Rerefinement with boost and concrete overtakes it from 2,000 steps on

**Learning**

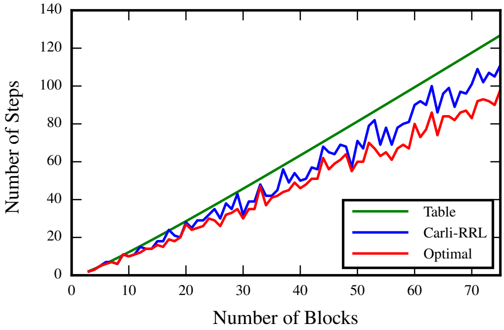



N – No rerefinement
U – Unrestricted rerefinement
B – Blacklists
O – bOost
C – boost with Concrete

# Policy Optimality Scaling

- The policy learned by my agents with a general solution to **`exact`** with variable target configurations, compared to
- Optimal calculated with $\mathrm{A}^*$ and
- An expected number of steps for a policy moving all blocks to the table and then into place

## As Blocks Increase



# Nuggets and Coal

## Nuggets

1. Successfully embedded an adaptive Hierarchical Tile Coding (aHTC) in a Rete
2. Demonstrated architectural flexibility using three different refinement criteria in addition to rerefinement, blacklist, boost, and concrete mechanisms
3. A general policy for `exact` that scales for arbitrary numbers of blocks

## Coal

1. Computational costs to execute policy are high for hundreds of blocks
2. Policy is only approximately optimal, but problem is NP-hard
3. I've been a student at U-M almost as long as Kenan Thompson has been a cast member of SNL

Hamid Reza Maei and Richard S Sutton.
Gq ($\lambda$): A general gradient algorithm for temporal-difference prediction learning with eligibility traces.
In *Proceedings of the Third Conference on Artificial General Intelligence*, volume 1, pages 91–96, 2010.