Algorithmic Human-Robot Interaction

Reinforcement Learning

CSCI 7000

Prof. Brad Hayes

Computer Science Department

University of Colorado Boulder

Final Papers

Due Tuesday at 11:59pm

- No presentation required – Online submission only

Mandatory Files

- Zip file titled “Paper.zip”
  - Final PDF
  - LaTeX / Word files
  - Figures
  - Link to a video presentation (up to 10min)
    - Narration over slides is preferred
- Zip file titled “Code.zip”
  - Checkout of your (up-to-date) Git repository
  - README file including instructions to run your project

Highly Encouraged

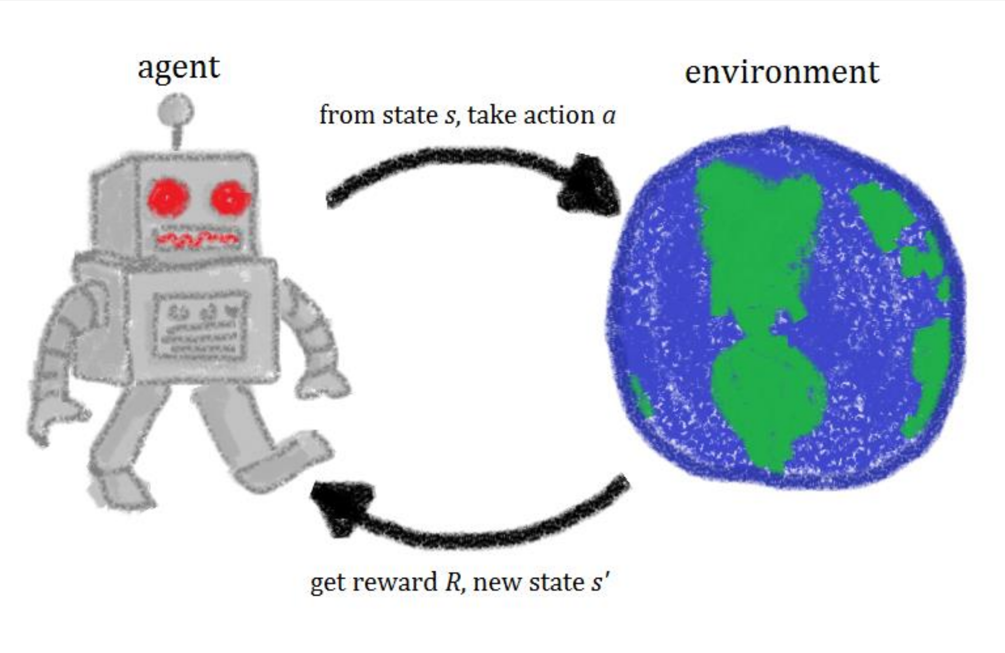
- Video demonstration of results in your presentation

Paper for Today:

A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning
by Ross et al.

Pro: Nishank Sharma
Con: Ashwin Vasan

Getting Started with RL

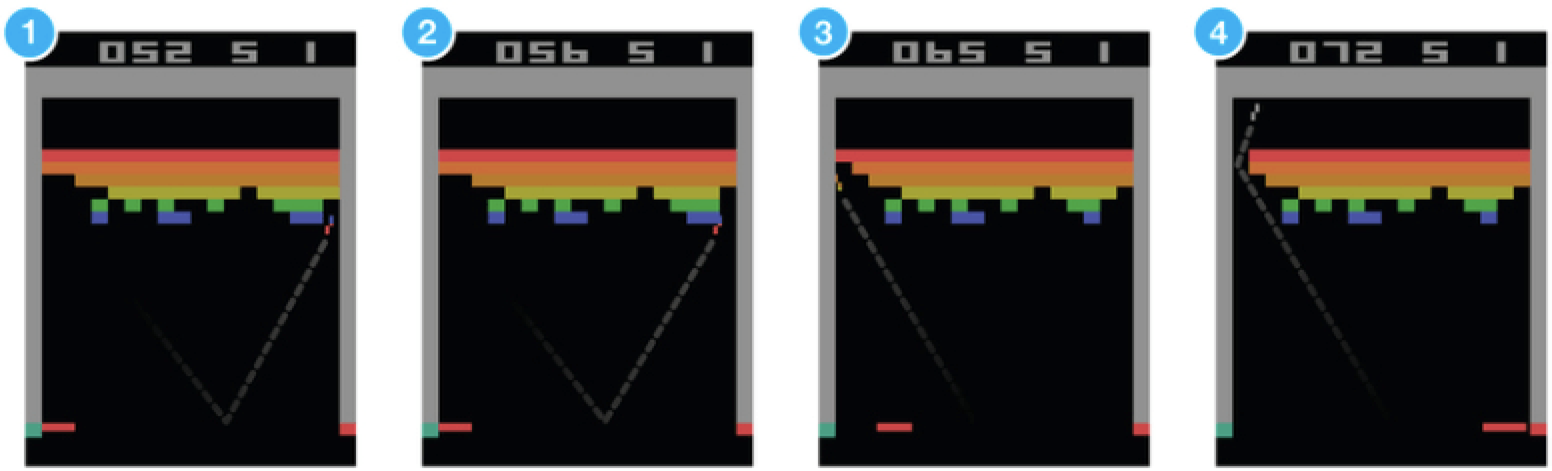


Prerequisites

Numpy:	pip install numpy
OpenAI Gym:	pip install gym
Torch:	pip install torch

Download the rl-demo.py source code on Moodle!

Reinforcement Learning



Unsupervised Learning

- No labels at all!
- Only given data (Clustering/embedding problem)

Reinforcement Learning

Sparse labeling
(Few states get a correct answer)

Time-delayed signaling
(Labels not provided in a timely manner)

Supervised Learning

Target label for every example
(each state has the 'correct' action)

No need for reward function!

Breakout:
State: Position of ball, paddle, bricks, etc.
Observation: Pixels from screen
Actions: Left, Right, Release Ball

Exploration / Exploitation

How often should you listen to your own strategy?

When should you give up acting randomly to find a new strategy?

A typical formulation is ϵ-greedy exploration:

Follow policy π most of the time (ϵ%), act randomly $(1 - \epsilon)$% of the time

More complex functions can also work well:

- Set an ϵ schedule (start value α, end value ω, decay δ)
- $\epsilon = \omega + \max(0, (\alpha - \omega)) * e^{-t/\delta}$

Discounted Future Reward

To perform well long-term, the agent needs to consider immediate rewards AND future rewards:

$$R = r_1 + r_2 + \cdots + r_n$$

But stochastic environments don't make r_{t+1} a certainty

The further into the future we look, the less certain we can be.

We address this by **discounting future reward**:

$$\boldsymbol{R = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n}$$

Discounted Future Reward → Q-Function

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^{n-t} r_n$$

$$R_t = r_t + \gamma\big(r_{t+1} + \gamma(r_{t+2} + \cdots)\big)$$

$$R_t = r_t + \gamma R_{t+1}$$

$$Q(s_t, a_t) = max\, R_{t+1}$$

Q tells us the best possible score at game-end if I do a_t in state s_t

$$\pi(s) = argmax_a\, Q(s, a)$$

Deriving the Q-Function

$$\boldsymbol{Q(s_t, a_t) = max\ R_{t+1}}$$

Q tells us the best possible score at game-end if I do $\boldsymbol{a_t}$ in state $\boldsymbol{s_t}$

$$\boldsymbol{\pi(s) = argmax_a\, Q(s, a)}$$

How do we derive Q?

Consider a single transition:

<s,a,r,s'> :: (state, action, reward, next state)

$$Q(s,a) = r + \gamma * \max_{a'} Q(s', a')$$

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

Q-Learning: Basic Algorithm

$$Q(s,a) = r + \gamma * \max_{a'} Q(s',a')$$

```
initialize Q[num_states,num_actions] arbitrarily
observe initial state s
repeat
      select and carry out an action a
      observe reward r and new state s'
      Q[s,a] = Q[s,a] + α(r + γ max_a' Q[s',a'] - Q[s,a])
      s = s'
until terminated
```

```python
  self.Q = np.random.random([NUM_BINS,NUM_BINS,NUM_BINS,NUM_BINS,2])  # Value of st


def select_action(self, state):
  sample = random.random()
  eps_threshold = self.EPS_END + (self.EPS_START - self.EPS_END)
                  * math.exp(-1. * self.steps_done / self.EPS_DECAY)
  self.steps_done += 1
  if sample > eps_threshold: # Exploit vs. Explore check
    return self.Q[tuple(state)].argmax()
  else:
    return random.randrange(2) # Pick random action

def update_model(self, state, action, reward, state_next):
  state_action_q_index = tuple(np.hstack([state, action]))
  max_future_q = self.Q[tuple(state_next)].max()
  self.Q[state_action_q_index] = self.Q[state_action_q_index] + self.LEARNING_RATE
      * (reward + self.GAMMA * max_future_q - self.Q[state_action_q_index])
```

Tabular Q-Learning Doesn't Scale

DeepMind used 4 grayscale 84x84 frames for their Nature paper
"Human-level control through deep reinforcement learning"

State Space Issues:

- 84x84 Pixel Screen * 4 frames @ 256 grayscale levels = $256^{84*84*4} \approx 10^{67970}$
- Q-Table would have to have 10^{67970} rows!
- Most states are never visited, others very rarely

Solution:

- Need a function approximator that's very good at learning features for structured data
- Deep Learning to the rescue

Deep Q-Network: Neural Net as Q-Function

Deep Q-Network Architecture

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

(Convolutional Neural Network architecture from Mnih et al.)

18 possible output actions

Deep Q-Network Architecture + Loss

Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

$$L = \frac{1}{2}[\underbrace{r + \max_{a'} Q(s', a')}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}}]^2$$

Experience Replay

- Deep networks approximate highly non-linear functions
- Updates at each time step are too similar
  - Drives network into a local minimum
  - Need to diversify updates to prevent this from happening
- Solution:
  - Store all <s,a,r,s’> tuples experienced during learning
  - Pick random batches of experiences to train with at each update step
  - Makes sure your network doesn’t “forget” previous information

Deep Q-Learning

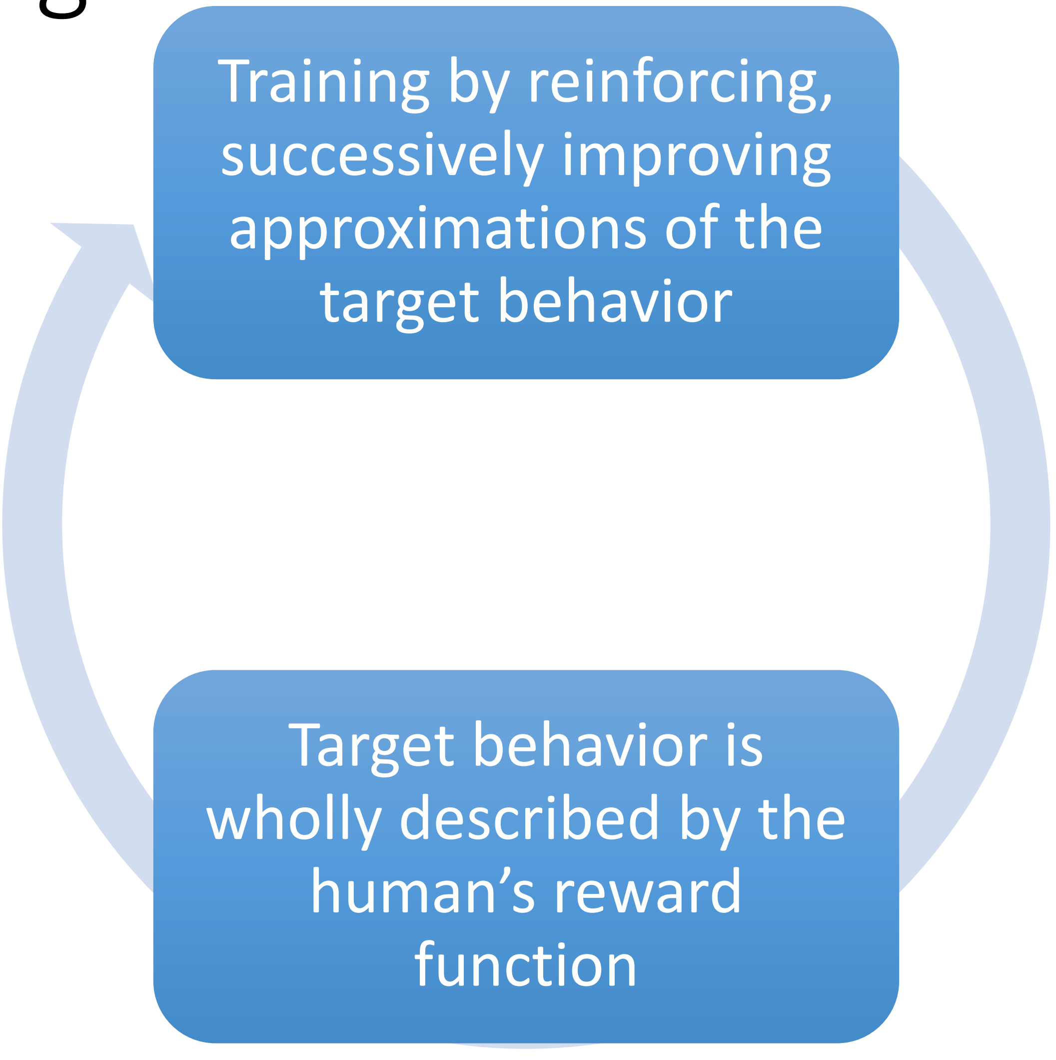
```
initialize replay memory D
initialize action-value function Q with random weights
observe initial state s
repeat
     select an action a
          with probability ε select a random action
          otherwise select a = argmax_a' Q(s,a')
     carry out action a
     observe reward r and new state s'
     store experience <s, a, r, s'> in replay memory D

     sample random transitions <ss, aa, rr, ss'> from replay memory D
     calculate target for each minibatch transition
          if ss' is terminal state then tt = rr
          otherwise tt = rr + γmax_a' Q(ss', aa')
     train the Q network using (tt - Q(ss, aa))^2 as loss

     s = s'
until terminated
```

Search is bad at forecasting

Shaping


Training by reinforcing, successively improving approximations of the target behavior
Target behavior is wholly described by the human's reward function

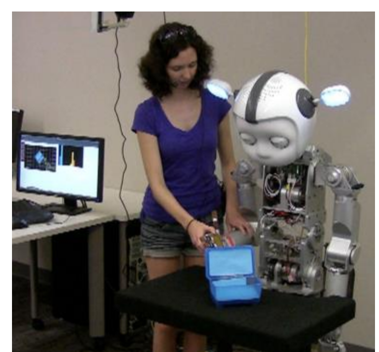
Why Bother "Shaping"?

- Decreases sample complexity
- Reduces or eliminates dependence on environment reward
- Doesn't require a domain expert

Shaping: Iteratively improving agent policy using a human reward signal

Related Work

- Learning from Advice
  - Works for people, but extremely complicated
- Learning from Demonstration
  - Human performance replaces standard environmental reward
  - Limited:
    - Human must be an expert
    - Not always a clear interface for the human to demonstrate with



TAMER Framework

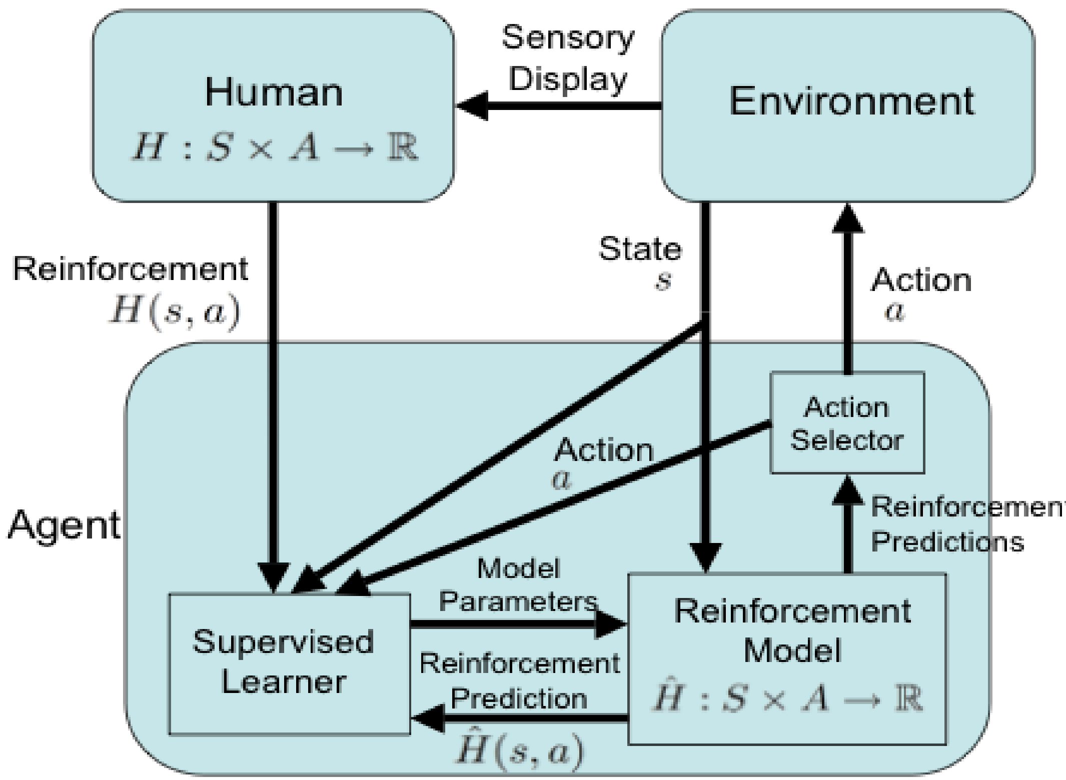


Figure 1: Framework for Training an Agent Manually via Evaluative Reinforcement (TAMER).

Advantages of Human Signal

Human automatically engages foresight

- Algorithm doesn't have to look ahead quite as far

Human can act in a tight temporal window

- Credit assignment can be frequent and be indicative of long term benefit to state-action pairs

Human reward is a moving target

- Human reinforcement will be more sparse over time
- Complacency causes agent to level performance

Challenges of Human Signal

Exploration

- Open research problem
- “To be filled by the agent designer”
- Experiments suggest that greedy selection provides sufficient exploration
  - Traditional RL maximizes return
    - Discounted sum of all future reward
  - TAMER maximizes immediate return without regard for future states

Exploration Continued

- TAMER acts in the present
  - Assumes the human is thinking ahead for the agent
  - Becomes problematic if sensors exceed human perception of the environment
    - Vehicle is moving too quickly for human to perceive nail in the road
      - Glint detector would detect object
    - UAV radar detects an inbound missile
      - Human can't react quickly enough to change agent behavior

Credit Assignment

- Time step frequency can exceed human capacity for response
  - Anything below 200ms is too fast
- Credit provided via simple algorithm:
  - Choose PDF (Gamma distribution)
    - Probability over time of particular time slice being targeted with reward
  - Proportion of credit equal to the integral over the execution window between timesteps.
    - Credit used to weight the correction term in Q-function updates

$$\Gamma(\theta,k)= x^{k-1}\frac{\exp\left(-x/\theta\right)}{\Gamma(k)\,\theta^{k}}$$

Credit Assignment

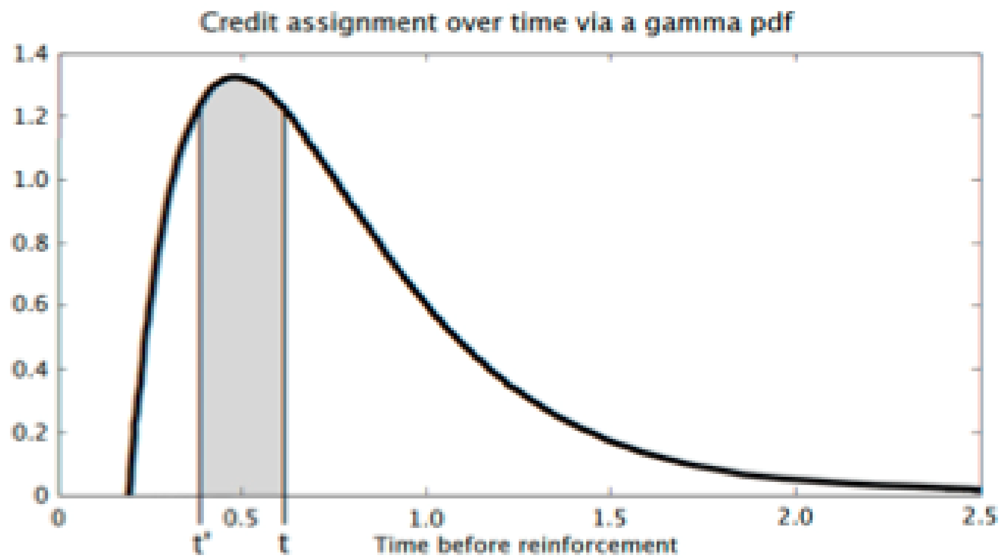


Figure 3: Probability density function $f(x)$ for a gamma(2.0, 0.28) distribution. Reinforcement signal h is received at time 0. If t and t' are times of consecutive time steps, credit for the time step at t is $\int_{t'}^{t} f(x)dx$. Note that time moves backwards as one moves right along the x-axis.

Example

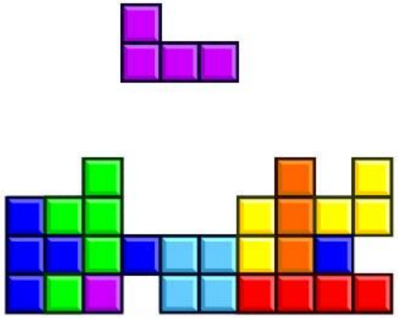
Algorithm 2 A greedy TAMER algorithm with credit assignment, using a linear model and gradient descent updates

---

Require: *Input: stepSize, windowSize,*

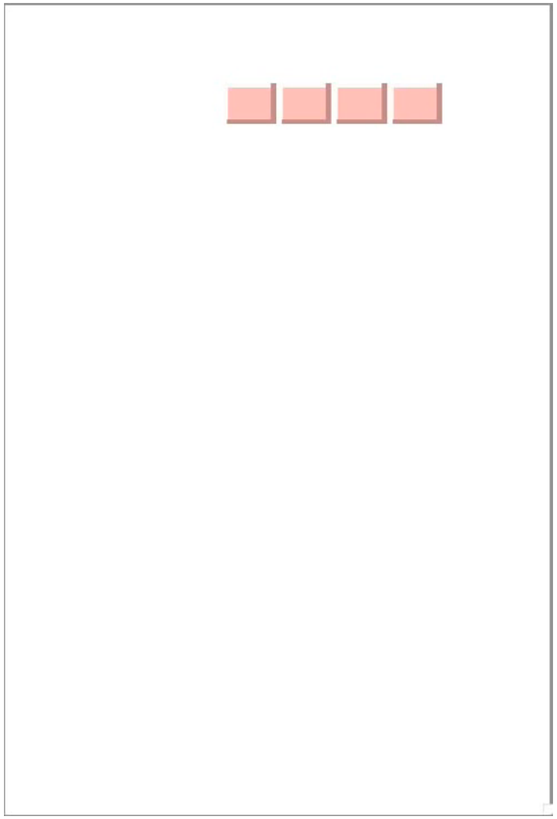
$Crediter.init(windowSize)$
$\overrightarrow{s} \leftarrow \overrightarrow{0}$
$\overrightarrow{f} \leftarrow \overrightarrow{0}$
$\overrightarrow{w} \leftarrow \overrightarrow{0}$
while *true* **do**
    $Crediter.updateTime(clockTime())$
    $h \leftarrow getHumanReinfSincePreviousTimeStep()$
    if $h \neq 0$ **then**
        $\overrightarrow{credFeats} \leftarrow 0$
        for all $(\overrightarrow{f_t}, t) \in Crediter.historyWindow$ **do**
            $c_t \leftarrow Crediter.assignCredit(t)$
            $\overrightarrow{credFeats} \leftarrow \overrightarrow{credFeats} + (c_t \times \overrightarrow{f_t})$
        end for
        $error \leftarrow h - (\overrightarrow{w} \cdot \overrightarrow{credFeats})$
        $\overrightarrow{w} \leftarrow \overrightarrow{w} + (stepSize \times error \times \overrightarrow{credFeats})$
    end if
    $\overrightarrow{s} \leftarrow getStateVec()$
    $a \leftarrow argmax_a(\overrightarrow{w} \cdot (getFeatures(\overrightarrow{s}, a)))$
    $\overrightarrow{f} \leftarrow getFeatures(\overrightarrow{s}, a)$
    $takeAction(a)$
    $Crediter.updateWindow(\overrightarrow{f})$
    wait for next time step
end while

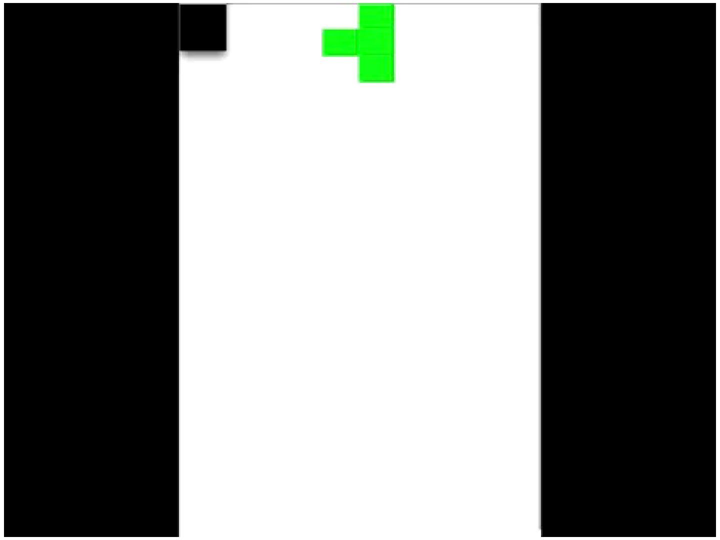
Example Domains: Tetris

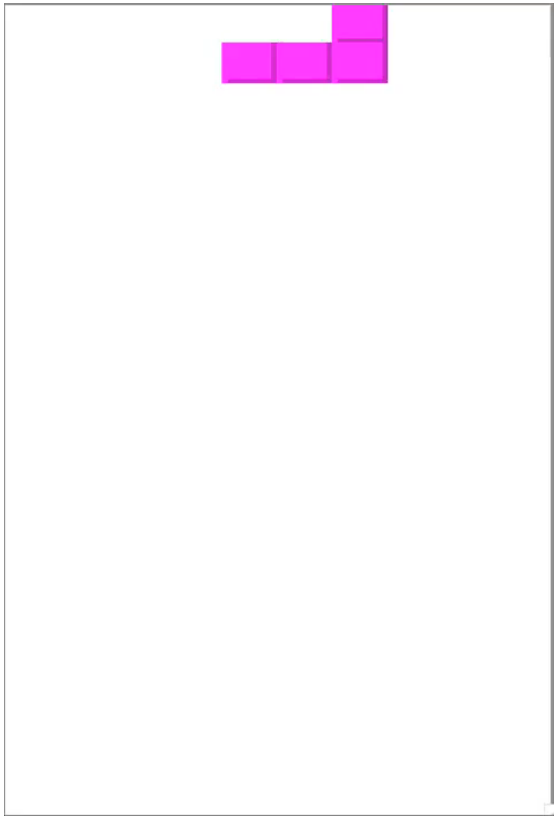


- Tetris
  - Low update frequency
  - Complex problem domain
    - Massive state space
    - TD-Learning struggles
      - Successful implementations cheat
  - Stochastic environment contributes to poor performance on other learners
  - TAMER learns ~65 lines per game in 3 training episodes
    - Policy search algorithms do much better, but require many , many more training episodes (~1000s of lines cleared)

TAMER in Action







Pre-Training

Training

Post-Training

Tetris

Table 1: Results of various Tetris agents.

Method	Mean Lines Cleared		Games
	at Game 3	at Peak	for Peak
TAMER	**65.89**	**65.89**	**3**
RRL-KBR [15]	5	50	120
Policy Iteration [2]	~ 0 (no learning until game 100)	3183	1500
Genetic Algorithm [5]	~ 0 (no learning until game 500)	586,103	3000
CE+RL [17]	~ 0 (no learning until game 100)	348,895	5000