Introduction to Reinforcement Learning

Scalable Computational Intelligence Group
Texas Advanced Computing Center
University of Texas at Austin

Amit Gupta
July 2024

Setup Environment

Run the following command to setup your environment and copy materials for lectures

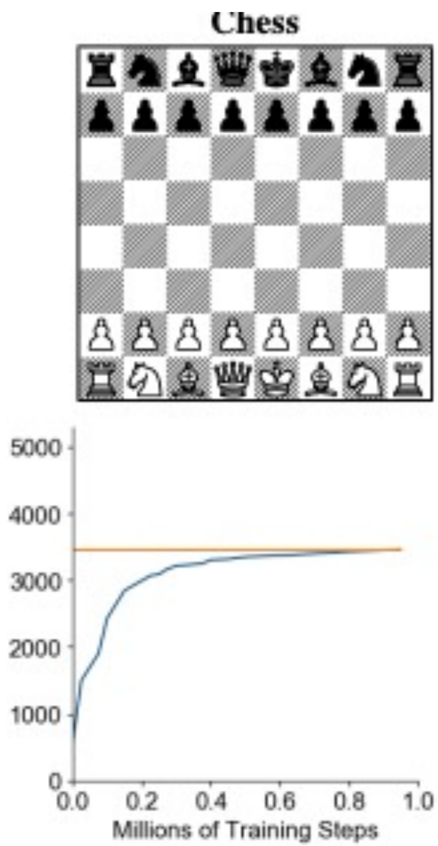
/scratch1/01596/jrduncan/ml_institute_setup/install

- Installs the container with the right Python and libraries for the training days
- Copies code for lectures into your home folder
  - **ml_institute_summer_24**
- Close your jupyter notebook session and relaunch

Introduction

- How should an agent behave in an environment to maximize its (cumulative) reward
  - A sub-field of Machine Learning
- Particularly useful in modeling games
- Framework for sequential decision making

Examples



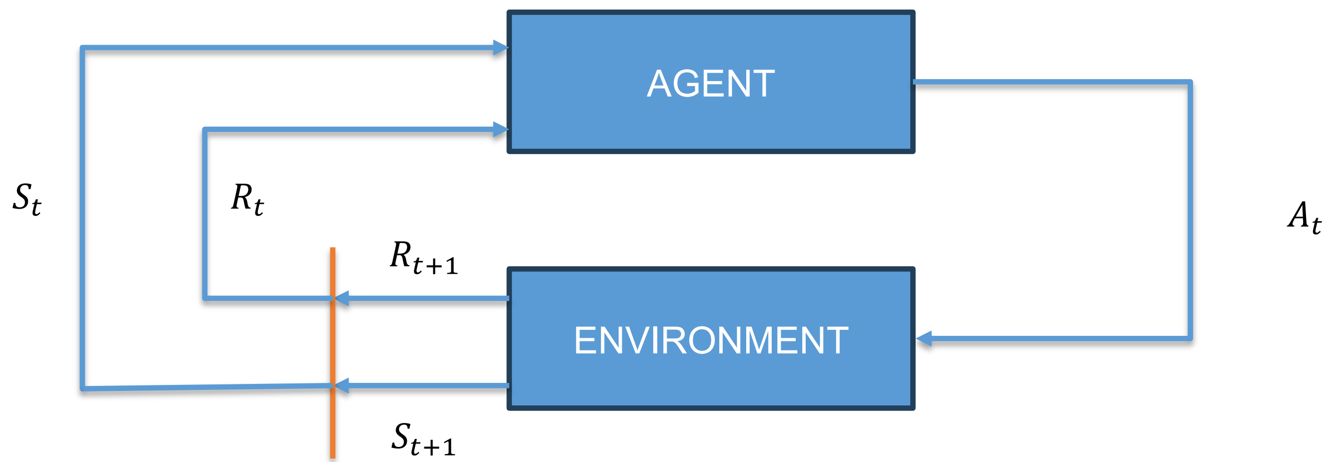
Deepmind: MuZero

Robotics Motor skills

Deepmind: Agent57

Markov Decision Process

- Agent
- Environment
- State
- Action
- Reward



Markov Decision Process

- State
  - $S_t \in \mathrm{S}$
- Action
  - $A_t \in \mathrm{A}$
- Reward
  - $R_t \in \mathrm{R}$
  - $R_{t+1} = f(S_t, A_t)$
- Trajectory
  - Sequence of events, starting at $t = 0$
  - $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2$ ...

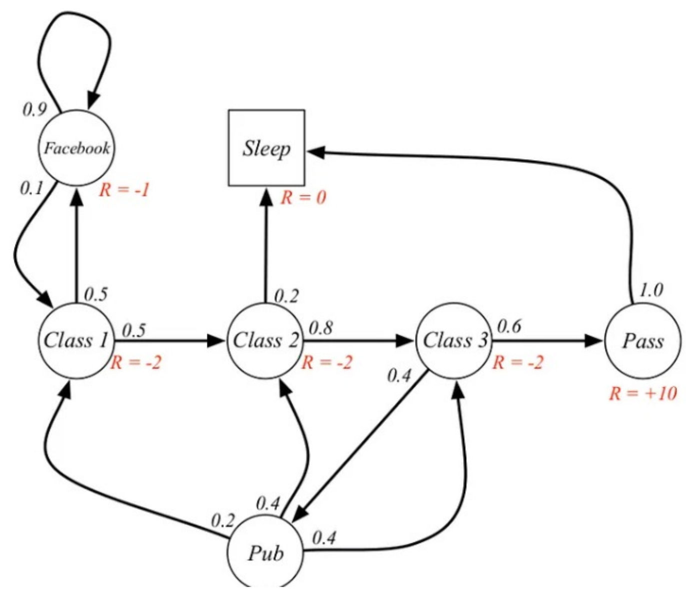
Markov Decision Process

- States follow the Markov property
  - The current state has all the information needed to predict the next state
    - Example an intermediate position in a chess game
  - $p(S_{t+1} \mid S_t) = p(S_{t+1} \mid S_1, S_2, \ldots S_t)$
- Current State and Reward S_t, R_t
  - Random variables sampled from the sets **S**, $\mathbf{R}$

TACC

Markov Decision Process

- State Transition Probability
  - Given state $s' \in S$ and reward $r \in \mathrm{R}$
  - Probability of $S_t = s'$ and $R_t = r$
    - Previous state $S_{t-1} = s$
    - Action taken $A_{t-1} = a \in A$
  - $p(s', r | s, a) = \Pr\{S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a\}$



Expected Return

- Objective of an agent in MDP
  - Maximize cumulative reward
- At any given time *t*
  - $G_t = R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T$
  - $T = final\ timestep$
- What happens in scenarios where $T = \infty$?
- Two types of RL tasks/games
  - Episodic Task
    - Theres a notion of a terminal state
    - Example: Single game in a set of ping-pong
    - MDP reset/restarts at initial condition
  - Continuous Task
    - They can go on forever, no terminal state
    - Example: Robot trying to balance while walking

Expected Return

- In continuous tasks where $T = \infty$
- Discount Rate $(0 < \gamma < 1)$
- Discounted Return (Modified Expected Return)
  - $G_t = R_{t+1} + \gamma . R_{t+2} + \gamma^2 . R_{t+3} + \cdots$
  - $G_t = \sum_{k=0}^{\infty} \gamma^k . R_{t+k}$
  - Makes infinite series converge
  - Places less importance on states that are far off
  - Another reduction
    - $G_t = R_{t+1} + \gamma . R_{t+2} + \gamma^2 . R_{t+3} + \cdots$
    - $G_t = R_{t+1} + \gamma . (R_{t+2} + \gamma^1 . R_{t+3} + \cdots)$
    - $G_t = R_{t+1} + \gamma . G_{t+1}$

Policy

- How should an agent select the next action?
- Policy Function
  - Maps current state to all possible actions that can be taken from that state
  - At any time t
    - Given State $S_{\mathrm{t}} = s \in S$
    - And given a possible Action $a \in A(s)$
    - $\pi(a|s) = \Pr(\mathrm{A_t} = \mathrm{a})$
      - Probability distribution over action space at state s
    - Agent is said to be following policy π at time t

TACC

Value Function

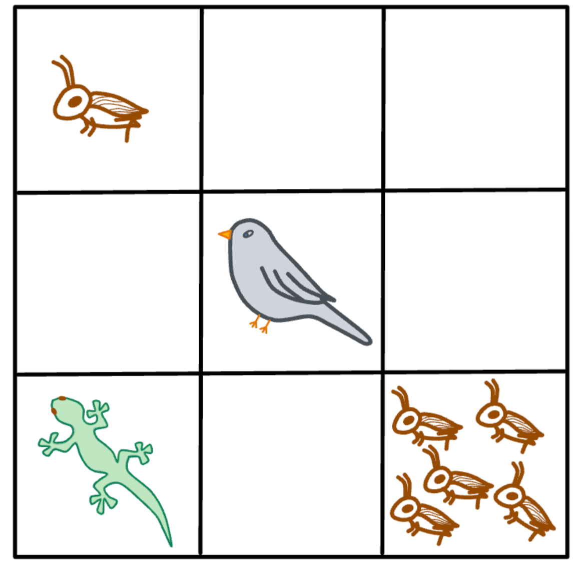
- How should an agent evaluate which action is better?
- Value Functions
  - State-Value
    - Value of the state in general
    - Expected return starting from state s and following π
    - $v_\pi(s) = E_\pi[G_t \mid S_t = s] = E_\pi[\sum_{k=0}^{\infty} \gamma^k . R_{t+k+1} \mid S_t = s]$
  - Action-Value
    - Value of taking a specific action a in state s when following π
    - $q_\pi(s,a) = E_\pi[G_t \mid S_t = s, A_t = a]$
    - $q_\pi(s,a) = E_\pi\ [\sum_{k=0}^{\infty} \gamma^k . R_{t+k+1} \mid S_t = s, A_t = a]$
    - Q – function q_π
    - Q – value $q_\pi(s,a)$

Optimal Policy

- Given a range of policies, which one is the best
  - $\pi \geq \pi' \; if \; and \; only \; if \; v_\pi(s) \geq v_{\pi'}(s) \;\; for \; all \; s \in S$
  - Optimal State-Value function
    - $v_*(s) = \max\big(v_\pi(s)\big) \; for \; all \; \pi$
  - Optimal Action-Value function
    - $q_*(s, a) = \max\big(q_\pi(s, a\,)\big) \, \forall \, \pi$
- Bellman Equation
  - $q_*(s, a) = E\big[R_{t+1} + \gamma . \max\big(q_*(s', a')\big)\big] \, \forall \, a'$
  - Optimal Q-value
    - Return from taking action a +
    - Discounted return of best possible action a' in next state s'

Lizard Game

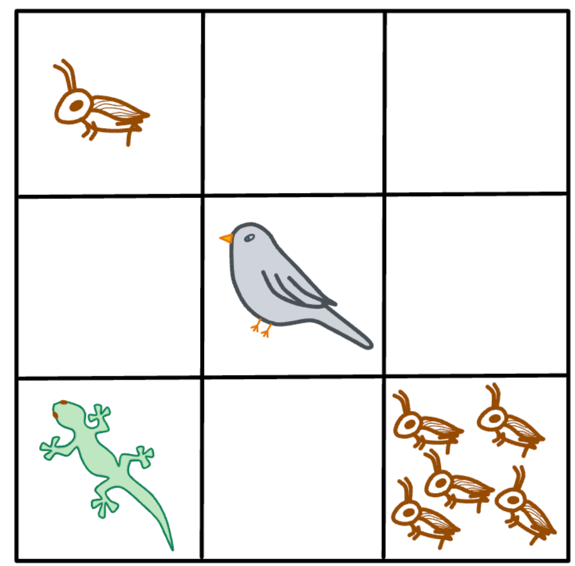
- Objective
  - Lizard eat as many crickets as possible
  - Lizard avoids the bird
- Action Space
  - Left
  - Right
  - Up
  - Down



Lizard Game

- Reward
- Environment
  - Grid
- State Space
  - Empty cell
    - 1
    - 2
    - 3
    - 4
    - 5
  - One Cricket
  - Bird
  - Five Crickets

State	Reward
One Cricket	+1
Empty	-1
Five Crickets	+10 (Game Over)
Bird	-10 (Game Over)



Q-table

Actions

States

	Left	Right	Up	Down
One Cricket	0	0	0	0
Empty 1	0	0	0	0
Empty 2	0	0	0	0
Empty 3	0	0	0	0
Bird	0	0	0	0
Empty 4	0	0	0	0
Empty 5	0	0	0	0
Empty 6	0	0	0	0
Five Crickets	0	0	0	0

Learning Q-values

- Q-table first initialized to zero
- Play several episodes of the game
  - Trial and Error approach
- In each episode
  - Calculate q-values $q(s, a)$
- Update Q-table based on these values and an update rule
- Referred to as value iteration
- Initially all entries are zero
  - All actions have same value
  - How to begin learning Q-values?

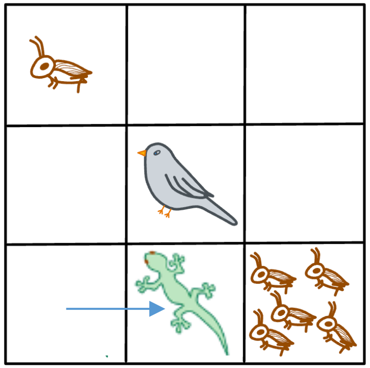
Exploration vs Exploitation

- Exploitation
  - Using existing knowledge of the environment learned from previous attempts in order to maximize return
- Exploration
  - Venturing into state/action pairs for which q-values are presently unknown
  - Helps agent learn new information about the environment
- A combination/tradeoff between both is required
  - Exploitation alone might cause agent to miss larger rewards
  - Exploration alone does not utilize information from previous attempts and effectively “re-learns” already known information

Exploration vs Exploitation

- Implemented using Epsilon Greedy Strategy
  - Hyperparameter $\in$ (Exploration Rate)
  - Initially $\in = 1$ (i.e 100% exploration)
- Exploration decay rate
  - Reduces $\in$ after each episode
  - Exploration probability reduces with successive attempts
- As Q-values get updated, they converge to the optimal values
  - $q^*(s,a) - q(s,a)$

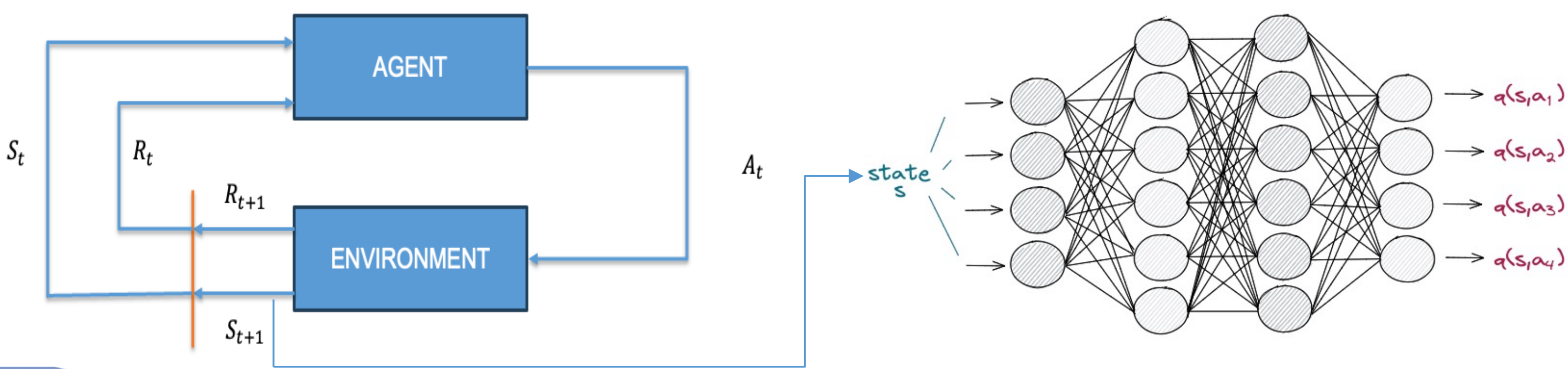
Exploration vs Exploitation



- Learning Rate α
  - Hyperparameter to balance
    - how much of old information is retained
    - how much new information is incorporated
    - $new\ q\ value = \ (1-\alpha)(old\ q\ value) + \ \alpha(learned\ return)$
  - $q_{new}(s,a) = (1-\alpha)q_{old}(s,a) + \alpha(R_{t+1} + \gamma_{a'}.\max(q(s',a'))$
- Example: If lizard first moves to the right
  - Let $\alpha = 0.7$, $\gamma = 0.99$
  - Empty cell, reward = -1
  - $q_{new}(s,a) = (1-\alpha)q_{old}(s,a) + \alpha(R_{t+1} + \gamma_{a'}.\max(q(s',a'))$
  - $q_{new}(s,a) =$ (1-0.7)(0) + 0.7(-1+(0.99)(0)) = -0.7
- Max timesteps can be specified to ensure termination

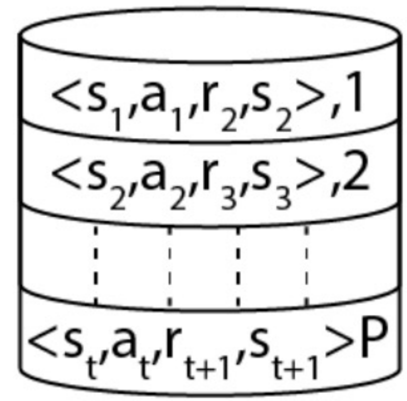
Deep Q-Learning

- Lizard game
  - Q-table dimensions 9 states x 4 actions
- For large state/action spaces
  - Maintaining Q-tables becomes inefficient
  - Alternative
    - Use a function approximator to learn $q_*(s, \mathrm{a})$



Deep Q-Learning
Replay Memory

- Experience Replay
  - Replay Memory
    - Agents experience stored in a dataset
    - Finite size (N)
    - $e_t = (s_t, a_t, r_{t+1})$
  - Randomly sample from memory
    - Batch Size
  - Train Neural Network



- Why use this?
  - Action in one state leads to another state
    - High correlation between “connected” states
  - Removes correlation between a sequence of state transitions. Avoids overfitting.

Deep Q-Learning
Training

- Neural Network Loss Function
  - Calculated by difference between
    - Current q-value (output by network)

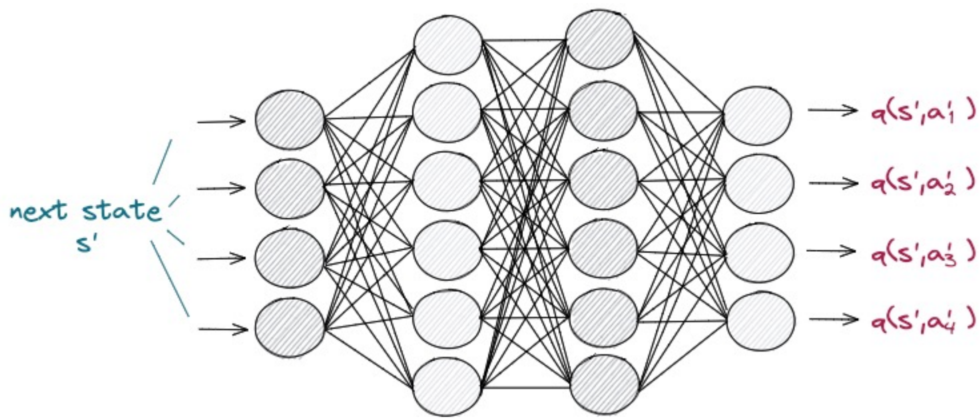
$$q(s,a) = E_{\pi}\left[\sum_{k=0}^{\infty} \gamma^{k}.R_{t+k+1}\right]$$

    - Target q-value (calculated using Bellman Equation)

$$q_{*(s,a)} = E[R_{t+1} + \gamma.\boxed{\max_{a'}(q_{*}(s',a'))}]$$

    - $loss = q_{*}(s,a) - q(s,a)$
- Calculating $\max_{a'}(q_{*}(s',a'))$
  - Requires another forward pass through the network for the next state s'

Deep Q-Learning
Training



- 1st pass: q-value of a state $q(s, a)$
- 2nd pass: Target q-value of that state $q_{*(s,a)}$
- When a parameter update is made both are now changed
  - $q(s, a)$ and $q_{*(s,a)}$ are both moving targets
  - Causes instability in the training

Target network

- Policy Network
  - Calculates Q-values for state-action pairs
- Target Network
  - Clone of the policy network
  - Weights stay frozen (until updated)
  - Calculates Target Q-Values for state-action pairs
  - ”Second pass”/next state is done here
    - Doesn’t change the weights of the policy network
  - Hyperparameter τ (target network update rate)
    - Copy over the policy network weights every τ time steps
- “Fixed” Target Q-values

TACC

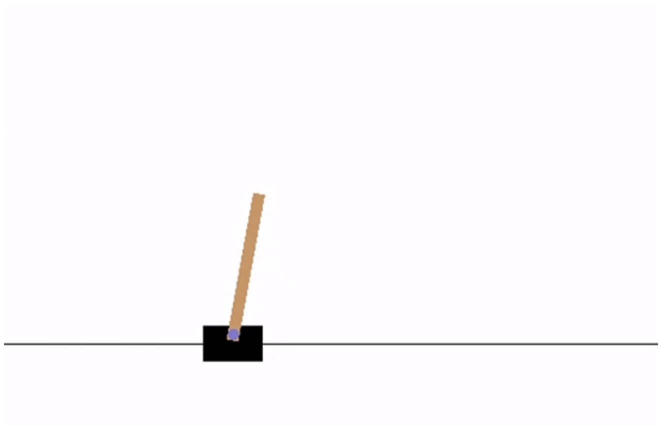
CartPole

- Move the cart on a frictionless surface such that pole stays balanced and within some angular limits
- Action Space
  - Move Left
  - Move Right
- State Space

	State	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-∞	∞
2	Pole Angle	-24°	+24°
3	Pole Angular Velocity	-∞	∞

CartPole

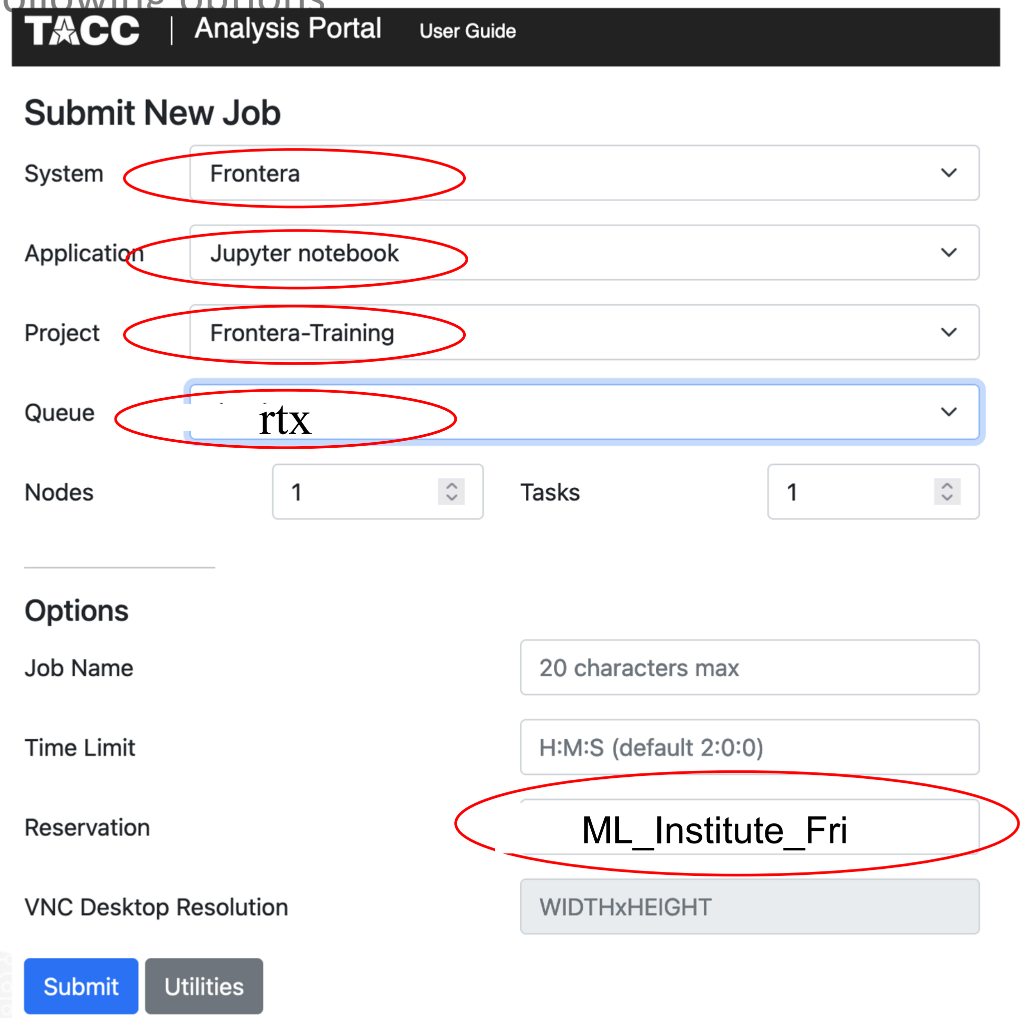
- Rewards
  - +1 for every step taken
- Starting Position of cart
  - Randomly sampled from (-0.05, 0.05)
- Episode Termination
  - Pole angle outside $\pm 12°$
  - Cart position outside ± 2.4
  - Max episodes set to 600



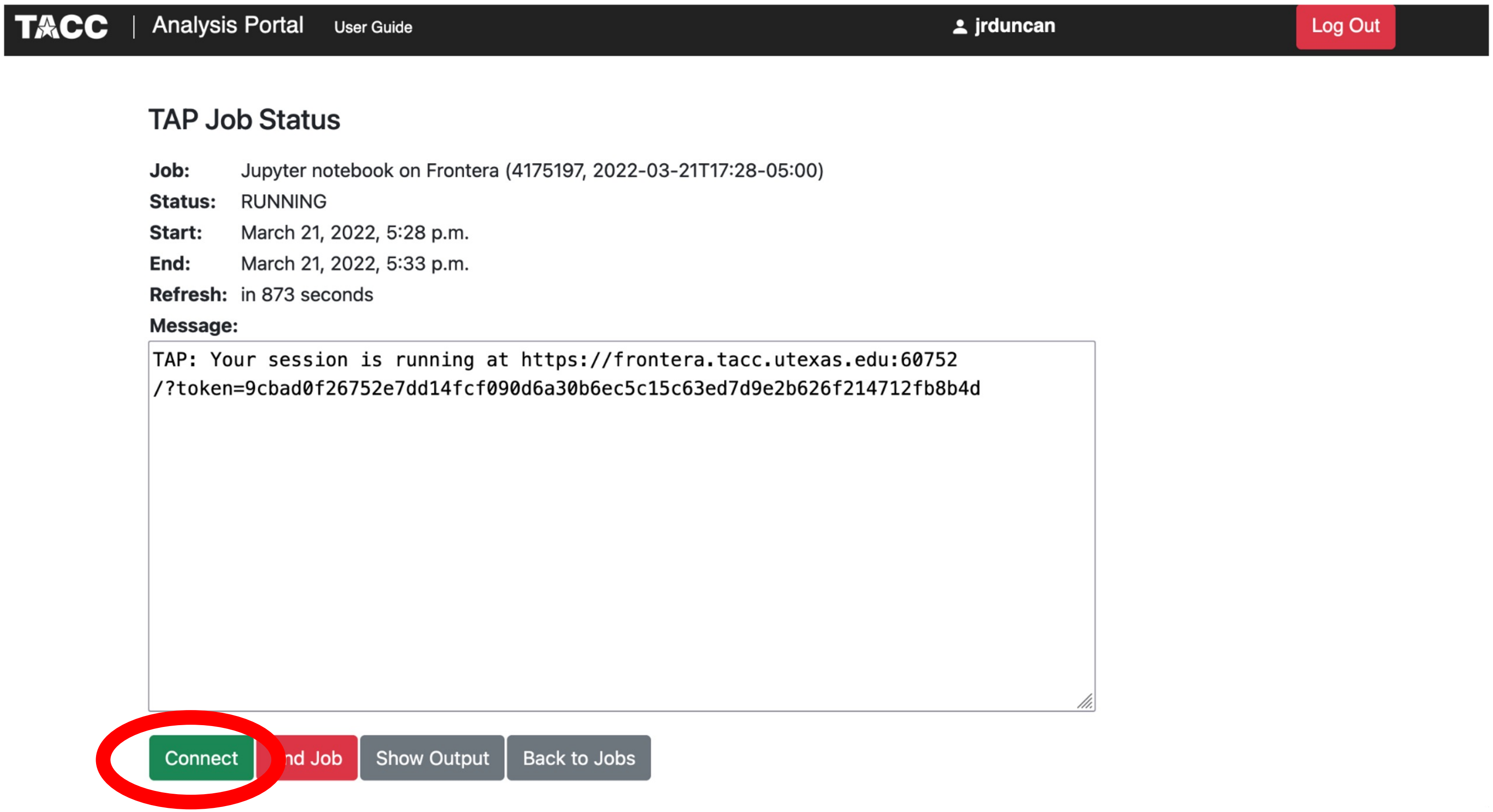
Jupyter notebook: Accessing shell

- TACC Analysis Portal: https://tap.tacc.utexas.edu

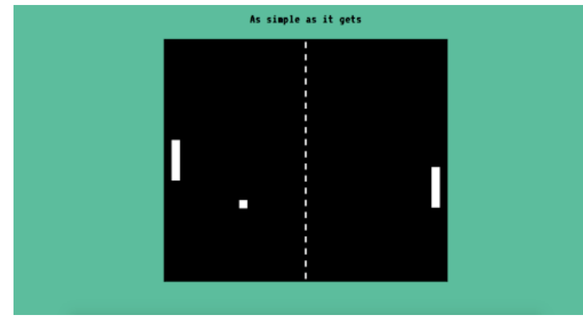
  Select the following options



Jupyter notebook: Accessing shell



Policy Gradient Method



- Optimise policy directly
  - $\pi_\theta(\mathrm{s}, \mathrm{a}) = \mathrm{P}(\mathrm{a}\,|\mathrm{s}, \theta)$
  - No value function
- Objective function
  - $\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s, a)\ Q^{\pi_\theta}(s, a)]$
  - Policy Gradient Theorem
- Possible advantages of this
  - Value functions can be complicated
  - Policy representation might be simpler
    - Easier to learn
  - Better convergence
- REINFORCE
  - Play out an episode
  - Take a sample of (state, action, reward)
  - Make policy parameter updates using above rule

Actor Critic Method

- Combines both policy based and value based methods
  - Value function approximation
  - Policy function approximation
- Two sets of parameters (i.e NN layer):
  - Critic (w)
    - Evaluating actors actions
    - Estimates action-value function $Q^w(s,a)$
  - Actor (θ)
    - Contains policy to choose actions
    - Updates policy parameters θ based on Critic estimates
- Objective function
  - $\nabla_\theta J(\theta) = E_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)\ Q^w(s,a)]$
  - Approximate policy gradient

References

- OpenAI Gymnasium module
  - https://gymnasium.farama.org/
- Reinforcement Learning: An Introduction
  - Sutton & Barto
  - http://www.incompleteideas.net/book/the-book.html
- DeepMind Reinforcement Learning Lectures
  - https://www.youtube.com/watch?v=2pWv7GOvuf0