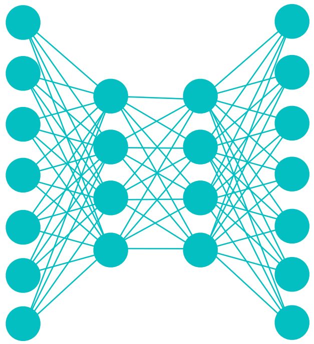
Lecture Notes for

Neural Networks and Machine Learning

Cross Entropy and Value Iteration

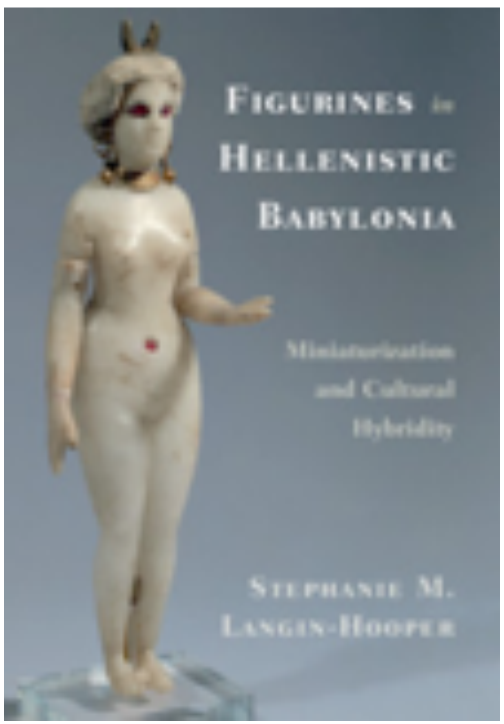


Logistics and Agenda

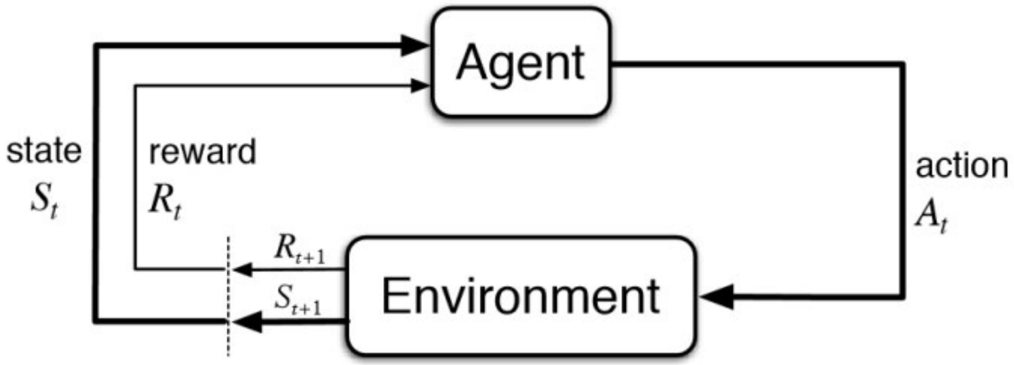
- Logistics
  - Atari paper next time!
  - Then, AlphaFold and SAC next week
- Agenda
  - OpenAI Gym
  - The Cross Entropy Method
  - Value Iteration
  - Q-Learning

Final Project

One Idea from Professor Stephanie Langin-Hooper
SMU Meadows

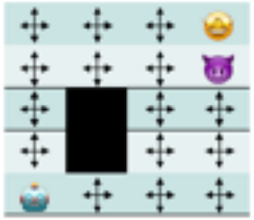


Last Time

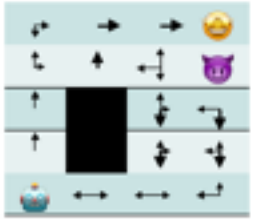


- **State**: Every square in grid
- **Action**: Move to make (l,r,u,d), with probability
- **Reward**: Goal, Death
- **Policy**: Given state, where should we move?
- **Optimal Policy**:

$$\pi^* = \arg\max_{\pi} \mathbf{E}\left[\sum_k \gamma^k R_{t+k+1} | \pi\right]$$


Random Policy


Another Policy

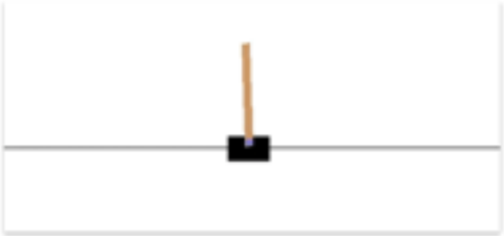

Another Policy

```python
import gym


if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break
```



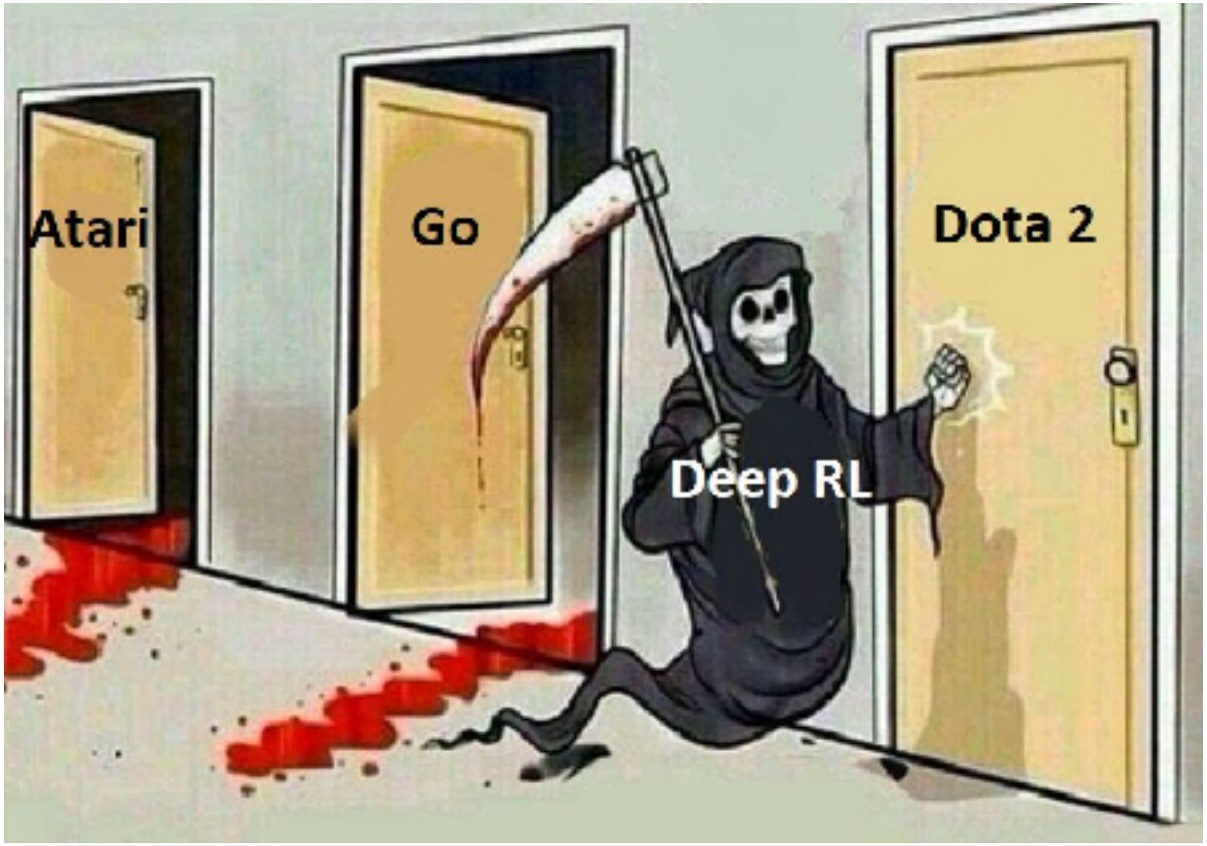
Action Space: One input, [0, 1] pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step

OpenAI Gym



Object Oriented RL

- Basics:
  - Define object instance for Agent() and the Env()
  - Define what observations will return
  - Run env.step(action)
  - Get new observations and reward from env
- `action_space` and `observation_space`
  - Possible actions to execute, Observations to get
  - Discrete or continuous?
  - Can actions be given simultaneously?

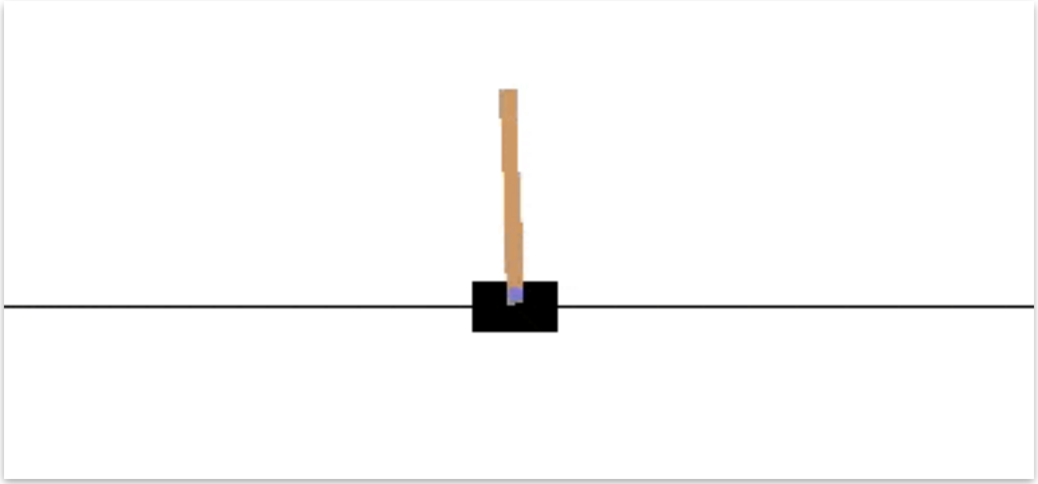
Basics of Cartpole

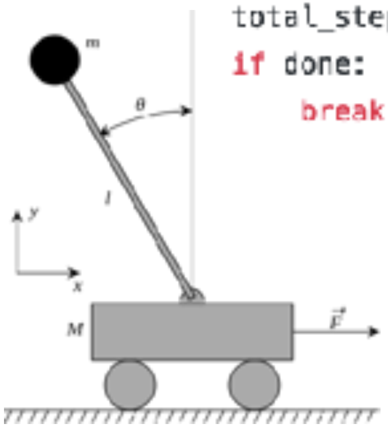
```python
import gym


if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break
```





Action Space: One input, [0, 1] pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step

Wrapping the Environment

- When you want some extra action, observation, reward processing
- Expose function with `ActionWrapper`, `RewardWrapper`, `ObservationWrapper`

```
class RandomActionWrapper(gym.ActionWrapper):
    def __init__(self, env, epsilon=0.1):
        super(RandomActionWrapper, self).__init__(env)
        self.epsilon = epsilon

    def action(self, action):
        if random.random() < self.epsilon:
            print("Random!")
            return self.env.action_space.sample()
        return action
```

```
if __name__ == "__main__":
    env = RandomActionWrapper(gym.make("CartPole-v0"))

    obs = env.reset()
    total_reward = 0.0

    while True:
        obs, reward, done, _ = env.step(0)
        total_reward += reward
        if done:
            break
```

Might return different action than user supplied with small probability

OpenAI Gym

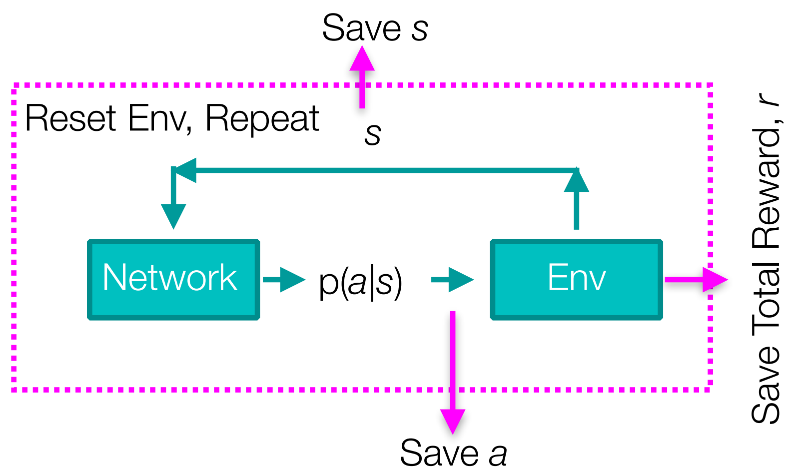
https://gym.openai.com

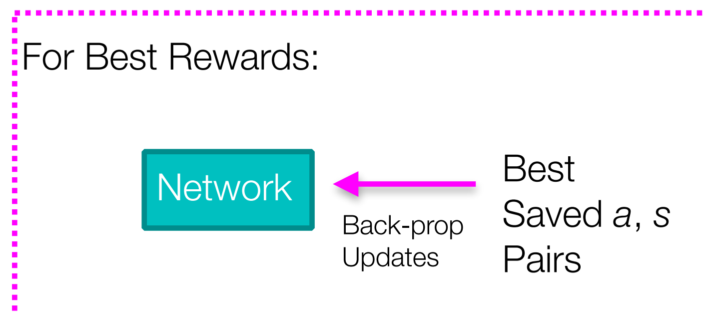
We provide the environment; you provide the algorithm.
You can write your agent using your existing numerical computation library, such as TensorFlow or Theano.

Cross Entropy Method

Optimize Best Random Models

- Create a random neural network
- Let it interact with the environment (randomly)
  - For some set of episodes (*e.g.*, 20)
    - Use network output to sample from possible actions
    - Run episode to completion
    - Repeat
- Calculate reward for each episode
- Keep best episodes (some percentile, e.g., best five)
- For the given best episodes, develop loss function incentivizing the actions taken based upon the input observations





Cross Entropy Method

- Model based or Model Free?
  - Model Free (no assumptions of problem)
- Value or Policy Based?
  - Policy Based (randomly sample actions based on policy)
- On-policy or Off-Policy?
  - On-Policy (need to interact with environment to get better)
- Has some similarity to **Simulated Annealing** Optimization

How to Make this More Mathy?

- If we have the optimal policy p(x) and a reward function H(x), then maximize

$$\mathbf{E}_{x \leftarrow p(x)}[H(x)] = \mathbf{E}_{x \leftarrow q(x)}[\frac{p(x)}{q(x)} H(x)]$$

- We can approximate the distribution by: $\frac{1}{N} \sum_i \frac{p(x_i)}{q(x_i)} H(x_i)$

- Proven that this is optimized when $\mathbf{KL}(\ q(x) \,||\, p(x)H(x)\ )$ is minimized. But its intractable, so we drop terms … and end up just minimizing (neg) cross entropy of samples

Performance Measure

$$\pi_{k+1}(a \,|\, s) = \arg\max_{\pi_k} \mathbf{E}_{z \leftarrow \pi_k}[\mathbf{1}_{R(z) > \psi} \log \pi_k(a \,|\, s)]$$

min CrossEntropy(*net_actions*, *best_actions*)

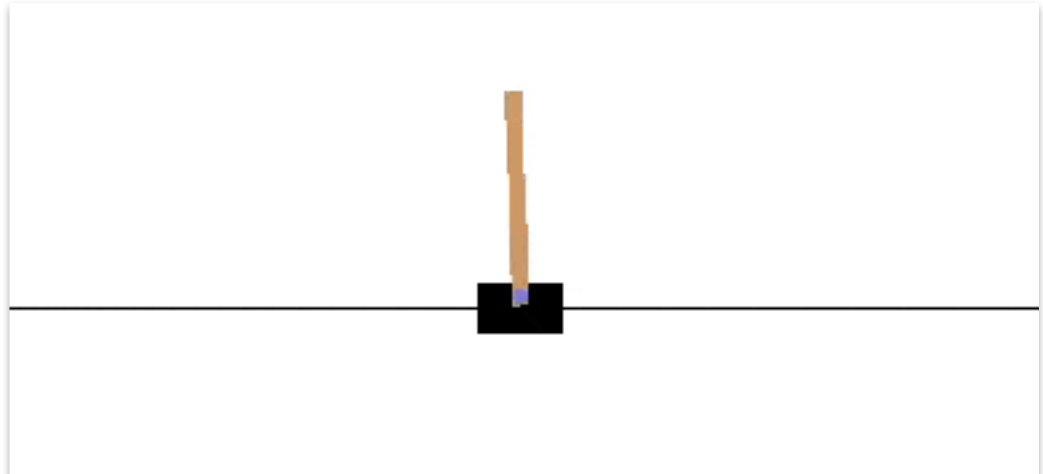
Review: Basics of Cartpole

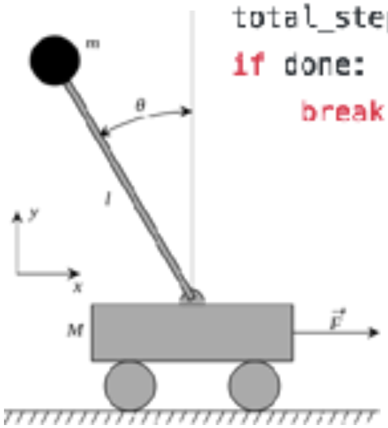
```python
import gym


if __name__ == "__main__":
    env = gym.make("CartPole-v0")

    total_reward = 0.0
    total_steps = 0
    obs = env.reset()

    while True:
        action = env.action_space.sample()
        obs, reward, done, _ = env.step(action)
        total_reward += reward
        total_steps += 1
        if done:
            break
```





Action Space: One input, [0, 1] pull left or pull right

Obs Space: Dynamic state variables (continuous and four dimensional)

End: When more than 15 degrees off or too far from center

Reward: +1 for each time step

Cross Entropy Reinforcement Learning

M. Lapan Implementation for CartPole and Frozen Lake

Follow Along:
`08a_Basics_Of_Reinforcement_Learning.ipynb`

Value Iteration

State Value Review

- Given: $$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \ldots = \sum_k \gamma^k R_{t+k+1}$$
- $V(s) = \mathbf{E}[G \mid s_t{=}s]$, expected Value of a given state over all future iterations
- **Important**: we can only calculate this exactly if we know:
  - all the rewards for all the states
  - the probabilities of transitioning to a given state from selecting an action
  - likelihood of successful action
  - Most of the time **we know none of this** when we approach the problem, because it assumes a model of the system

The Bellman Equation

- For the case when each action is successful and state is discrete, current V is easy to calculate:

$$V_0 = \max_{a \in 1 \ldots A} (r_a + \gamma V_a)$$

current value is immediate reward plus value of next state with highest value

- Which feels like cheating because we assume we know V_a … just go with it for now
- General extension for when actions are probabilistic:

$$V_0 = \max_{a \in A} \mathbf{E}[r_{s,a} + \gamma V_s] = \max_{a \in A} \sum_{s \in S} p_{a,0 \to s} \cdot (r_{s,a} + \gamma V_s)$$

-probabilities of getting to next state x (current value is immediate reward plus value of next state)
-$p_{a,0 \to s}$ probability of getting to state s from state 0, given that you perform action a

- To select action with best value we need reward matrix, $r_{s,a}$ and action transition matrix $p_{a,0 \to s}$

Defining the Q-Function

$$V_0 = \max_{a \in A} \mathbf{E}[r_{s,a} + \gamma V_s] = \max_{a \in A} \sum_{s \in S} p_{a,0 \to s} \cdot (r_{s,a} + \gamma V_s)$$

- Define intermediate function Q

$$Q(s, a) = \sum_{s' \in S} p_{a,s \to s'} \cdot (r_{s,a} + \gamma V_{s'})$$

- With some nice properties/relations:

$$V_s = \max_{a \in A} Q(s, a)$$

$$Q(s, a) = r_{s,a} + \gamma \max_{a' \in A} Q(s', a')$$

Value Iteration (Value Based)

- **Direct**:
  - Initialize V(s) to all zeros
  - Take a series of random steps
  - Perform for each state: $V(s) \leftarrow \max_{a \in A} \sum_{s' \in S} p_{a,s \to s'} \cdot (r_{s,a} + \gamma V(s'))$
  - Repeat until V(s) stops changing
- **Q-Function Variant**:
  - Initialize Q(s,a) to all zeros
  - Take a series of random steps
  - For each state and action: $Q(s,a) \leftarrow \sum_{s' \in S} p_{a,s \to s'} \cdot (r_{s,a} + \gamma \max_{a'} Q(s',a'))$
  - Repeat until Q is not changing

Need to estimate $p_{a,s \to s'}$
Via observed **Transitions**

This Update Will **Converge to Optimal Policy**

Value Iteration Reinforcement Learning

M. Lapan Implementation for
and Frozen Lake

Follow Along:
`08a_Basics_Of_Reinforcement_Learning.ipynb`

Some Limitations

- Q function can get really big for **large states and action spaces**
- Infinite when the spaces are **continuous**
    - We will solve this by using a neural network to **approximate** the Q function
- Transition matrix, similarly, can get gigantic for large state and action spaces
    - We will solve this by dropping the transition probabilities in Q function update
- This Variant is known as Q-Learning

Lecture Notes for

Neural Networks and Machine Learning

CE and Value Iteration

Next Time:
Deep Q-Learning

Reading: Lapan CH6, CH7