Playing Atari with Deep Reinforcement Learning

Mnih et al., 2013

Reinforcement Learning

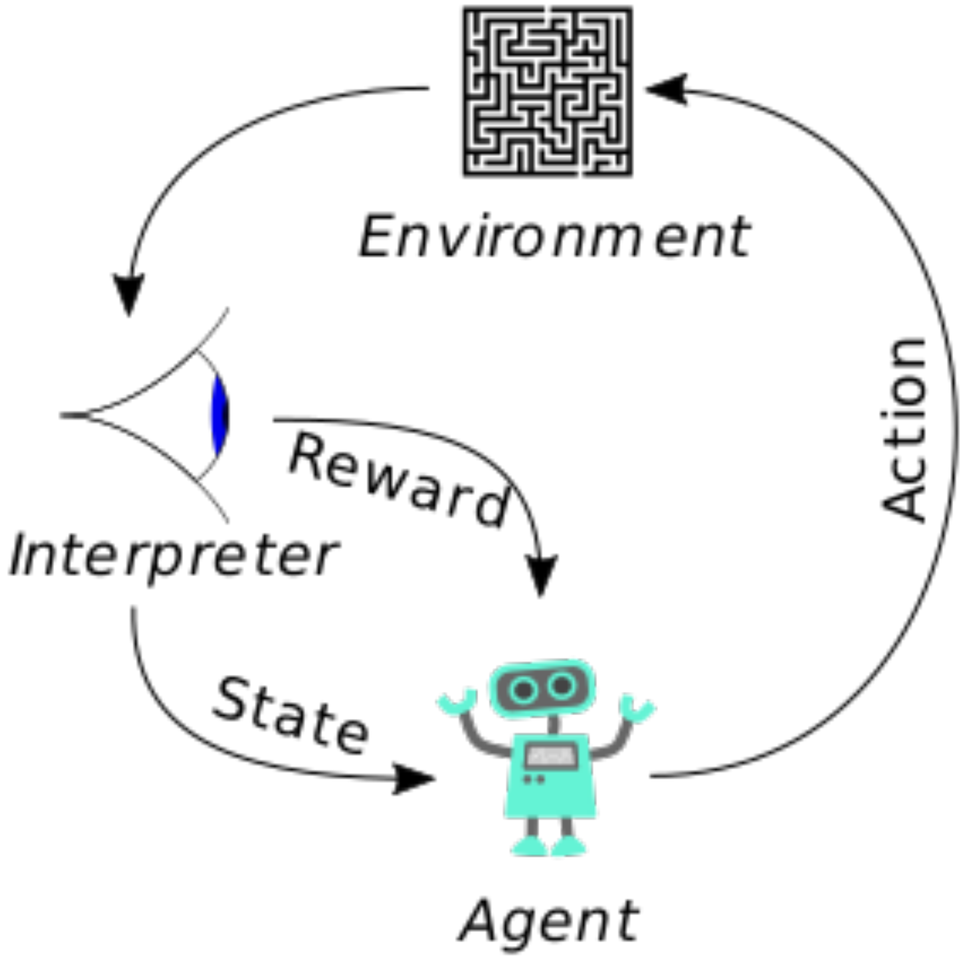
- Origins go back to 1957 with the Bellman equation/dynamic programming
- Still heavily researched, with several new papers each week on arXiv
- Learn actions from rewards, choose actions to maximize future rewards
- Agents can learn from direct interaction, simulation, or human replay
- Applications: elevator control, recommendations, robotics, finance, …

Reinforcement Learning Model

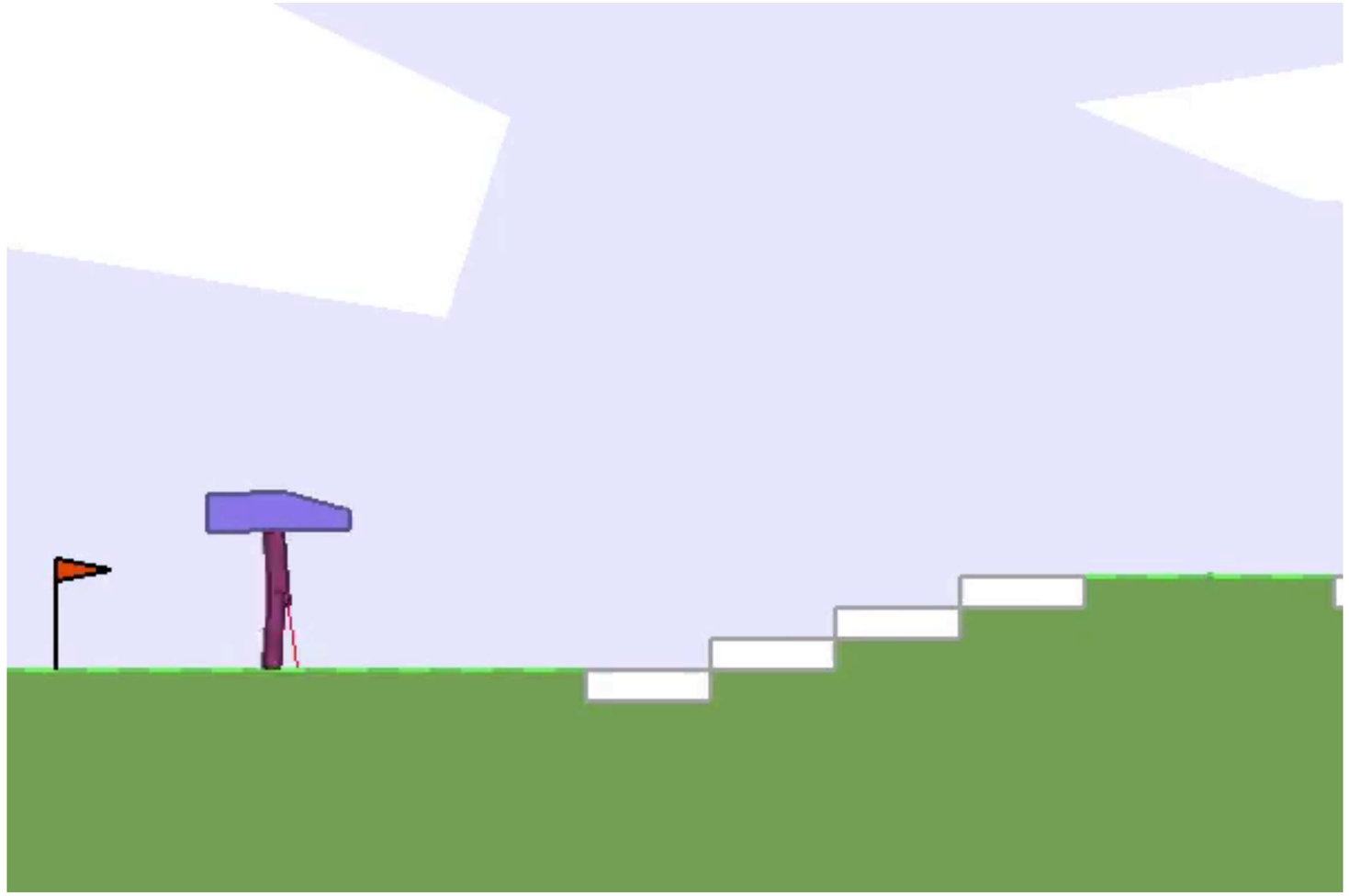
- **States** S, possibly infinite
- **Actions** A
- **Rewards** R
- **Policies** π: S x A -> [0, 1]

For finite state problems, the optimal policy can be found deterministically

Large or infinite state problems require heuristics and/or approximation

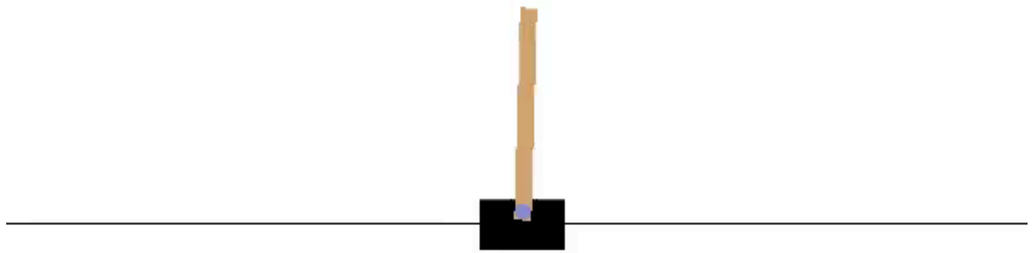


OpenAI Gym



Reinforcement Learning Hello World

- Balance a cart pole on a frictionless track for 200 steps
- State: position and angle of the cart
- Action: 1 unit of force on the left or right of the cart
- Reward: 1 for each step of survival



Exploration and Exploitation

- **Random** policy: choose a random action in every state (exploration)
- **Greedy** policy: choose the policy's best action (exploitation)
- **ε-Greedy** policy: with probability ε choose a random action, otherwise choose the policy's best action, decrease ε over time

EE is modeled more generally as the **multi-armed bandit** problem: choose the best action from a set of actions while learning the best action.

Deep Q-Learning Networks

- A function Q: S x A -> R models action rewards directly, without any underlying knowledge or assumptions about the environment
- Q represents the future reward for taking an action while in a given state
- For large state spaces, Q can be approximated using a neural network
- The network is trained online as the agent interacts with the environment

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

DQN Training

- DQNs basically perform linear regression, with a reward output for each action
- The output with the highest reward is the action taken by the policy
- Training minibatches are chosen from a replay memory, which maintains a sliding window over a history of state/action/reward/state tuples
- The replay memory ensures independence of samples, an assumption for neural networks

DQN Loss Function

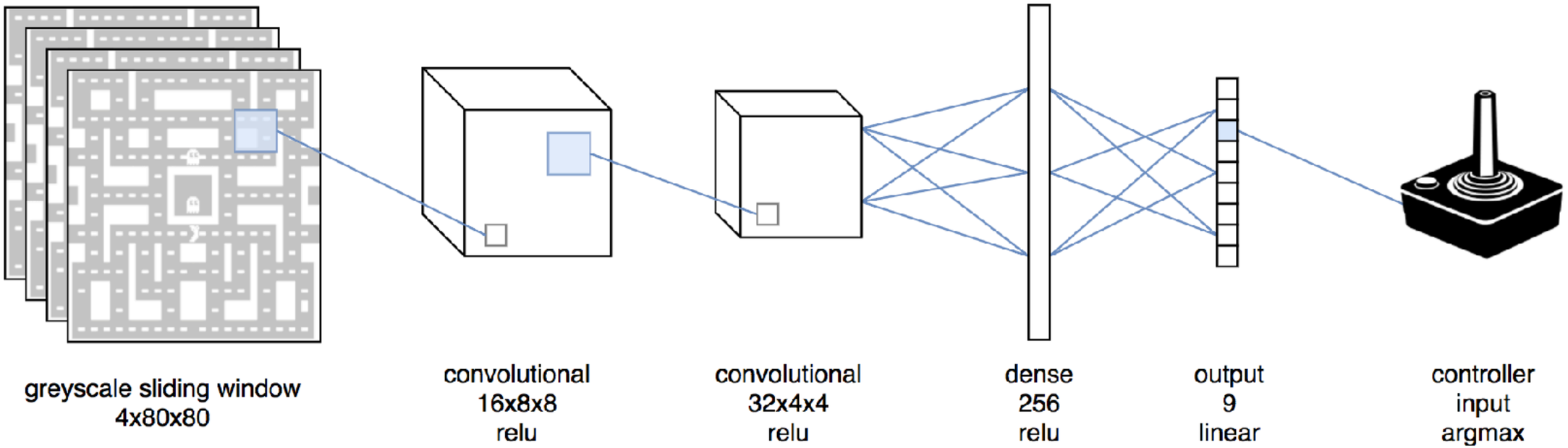
- The Q-Network performs simultaneous regression across all actions, so it needs a custom loss function

- The loss function calculates squared errors only for the winning action in the training sample

$$loss = (Y_{train} - Y_{output})[argmax(Y_{train})]^2$$

$$Y_{train} = [1, 2, 3, 2]$$
$$Y_{output} = [2, 4, 6, 8]$$
$$loss = (3 - 6)^2$$

Deep Q-Learning for Atari



- States: greyscale frame pixels (more than 2^200,000 states)
- Actions: controller inputs
- Rewards: points, survival, etc.

At an average score of 1000 on Ms. Pac-Man, a reward is received only once every 10 frames

Challenges

- Rewards are sparse and distant from causes
- Some action spaces are continuous or large
- Choosing model hyperparameters is done by hand
- There are newer and better methods than DQN (policy gradients, etc.)
- Google patented Deep Reinforcement Learning