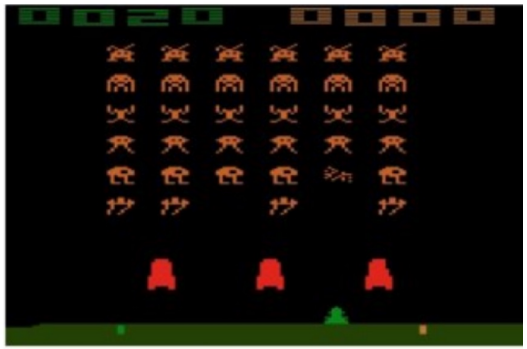
Lecture 15: Value-Based Deep Reinforcement Learning

DQN Demo

Q-learning:

- **Repeat**
  - Choose a for the state s (ϵ-greedy with $\arg\max_a Q(s, a)$)
  - Take action a, observe r and s'
  - Update:

$$\begin{aligned} Q(s, a) &\leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \\ s &\leftarrow s' \end{aligned}$$



- Difficult to represent $Q(s, a)$ as a lookup table when the number of states is large.

Deep Q-Network (DQN)

The Moving Target Problem

DQN with Target Network

Repeat:

- Take action a in current state s, observe r and s'
- Update the parameters:

$$\theta \leftarrow \theta - \alpha \nabla_\theta ([r(s,a) + \gamma \max_{a'} Q(s', a'; \theta^-)] - Q(s, a; \theta))^2$$

$$s \leftarrow s'$$

- $\theta^- \leftarrow \theta$ in every C steps.

DQN with Experience Replay

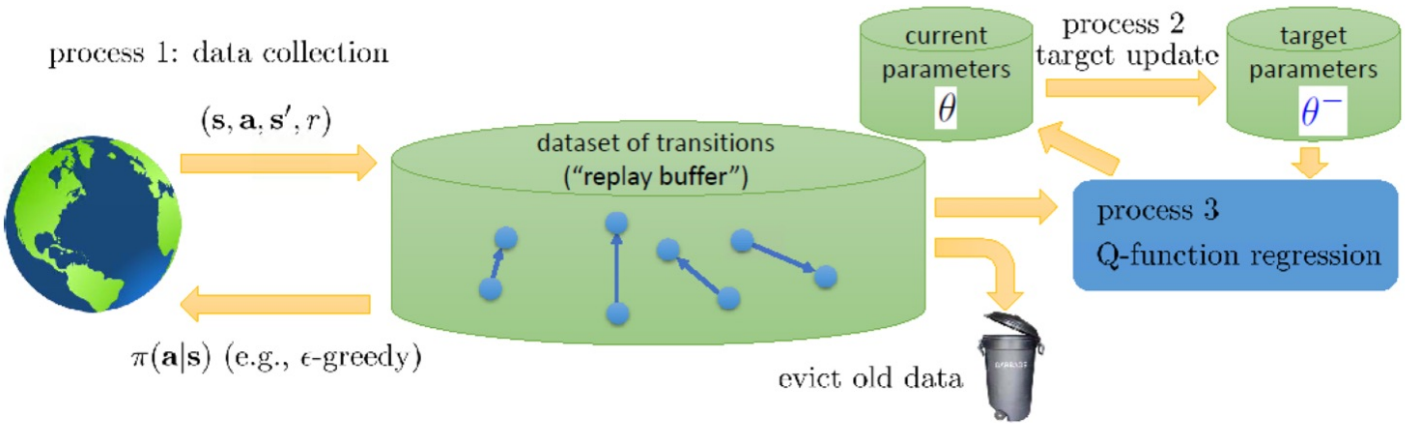
Repeat:

- Take action a in current state s, observe r and s'; add experience tuple (s, a, s', r) to a buffer D; $s \leftarrow s'$
- Sample a minibatch $B = \{s_j, a_j, s'_j, r_j\}$ from D.
- Update the parameters

$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_j ([r(s_j, a_j) + \gamma \max_{a'_j} Q(s'_j, a'_j; \theta^-)] - Q(s_j, a_j; \theta))^2$$

- $\theta^- \leftarrow \theta$ in every C steps.

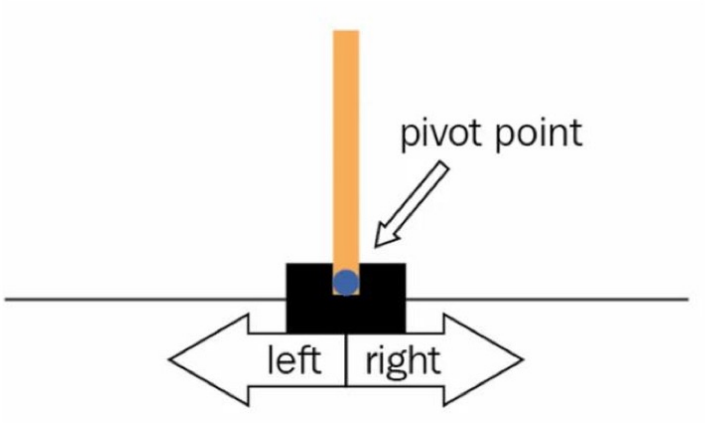
A More General View of DQN



- Processes 1 and 3 run at the same pace, while Process 2 is slower.
- Old experiences are discarded when buffer is full.

Tutorial 9 The Cart-Pole Problem

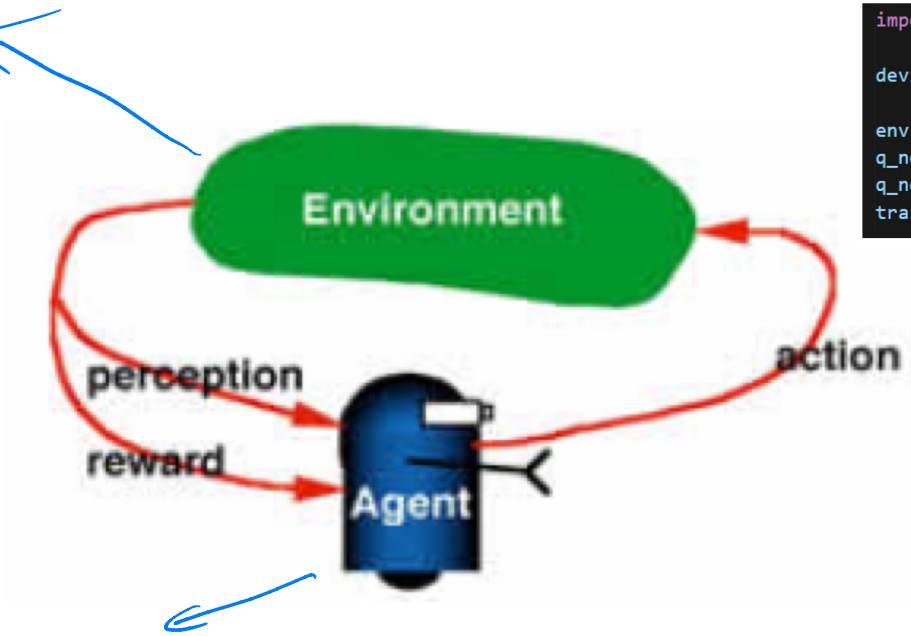
- A pole is attached to a cart that moves along a frictionless track.
- The objective is to balance the pole upright by applying forces to the cart, either left or right.



- **State Space**: the cart's position, velocity, the angle of the pole, and the angular velocity.
- **Actions**: push the cart left or push it right.
- **Rewards**: The agent receives a reward of +1 for every time step the pole remains upright. The episode ends when the pole falls beyond a certain angle or the cart moves too far from the center.

Demo : Cartpole-5000, Cartpole-1000, Cartpole-100

Gymnasium: API for RL, with reference environments included




```python
import gymnasium as gym

device = "cuda" if torch.cuda.is_available() else "cpu"

env = gym.make("CartPole-v1", render_mode="rgb_array")
q_net = DQN(4, 2)
q_net.to(device)
trainer = DQNTrainer(q_net, env)
```

DQN

Cart-Pole DQN

- DQN, replay buffer, ε-greedy

- DQN Trainer

  - Train : side-by-side with algo

  - update

Self.target_net

$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_j \left(\left[r(s_j, a_j) + \gamma \max_{a'_j} Q(s'_j, a'_j; \theta^-) \right] - Q(s_j, a_j; \theta) \right)^2$$

expected-state-action-vals

state-action-vals

Repeat:

- Take action a in current state s, observe r and s'; add experience tuple (s, a, s', r) to a buffer D; $s \leftarrow s'$
- Sample a minibatch $B = \{s_j, a_j, s'_j, r_j\}$ from D.
- Update the parameters

$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_j ([r(s_j, a_j) + \gamma \max_{a'_j} Q(s'_j, a'_j; \theta^-)] - Q(s_j, a_j; \theta))^2$$

- $\theta^- \leftarrow \theta$ in every C steps.

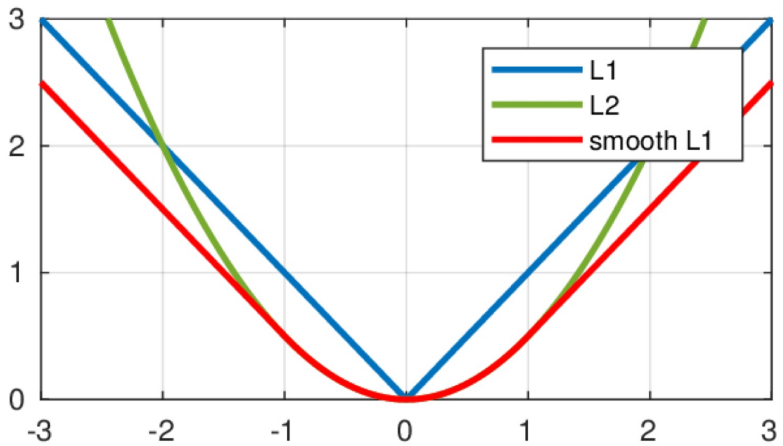
```
with torch.no_grad():
    next_state_vals[non_final_mask] = self.target_net.v_vals(batch.next_state)
#Q(s,a)=r + gamma*V(s')
expected_state_action_vals = batch.reward + self.gamma * next_state_vals

#compute loss & update
loss = self.criterion(state_action_vals, expected_state_action_vals)
self.opt.zero_grad()
loss.backward()
for para in self.q_net.parameters():
    para.grad.data.clamp_(-1, 1)
self.opt.step()
```

Instead of L2, it uses Smooth L1 Loss

$$\text{SmoothL1Loss}(x, y) = \begin{cases} 0.5 \cdot \frac{(x-y)^2}{\beta} & \text{if } |x-y| < \beta \\ |x-y| - 0.5 \cdot \beta & \text{otherwise} \end{cases}$$

Why Use SmoothL1Loss in DQN?

1. **Stability**: It helps stabilize training by reducing the sensitivity to outliers compared to Mean Squared Error (MSE) loss. This is particularly useful in reinforcement learning, where the temporal difference errors can vary significantly.
2. **Gradient Behavior**: The smooth transition between L1 and L2 loss allows for more stable gradients, which can prevent issues like exploding gradients during training.
3. **Performance**: Empirical results have shown that using SmoothL1Loss can lead to better convergence properties in DQN, improving the overall performance of the agent in learning optimal policies.



Time permitting ...

* Prioritized Experience Replay
* Double DQN
* RAINBOW

Lecture 16: Policy-Based Deep Reinforcement Learning

Introduction

Decision process with $\pi_\theta(a|s)$

$\nabla_\theta J$

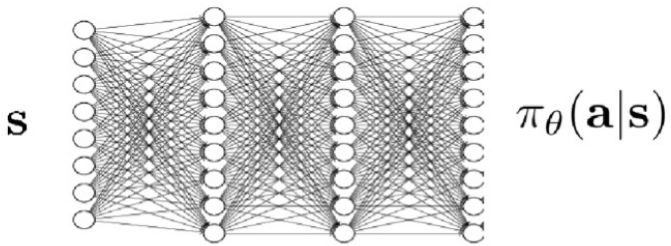
The REINFORCE Algorithm

- REINFORCE algorithm (Williams 1992):

  Repeat:

  1. sample $\{\tau^i\}_{i=1}^N$ from $\pi_\theta(a_t|s_t)$ (run the current policy)
  2. $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \sum_t \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) r(\tau^i)$
  3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

- The term $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is calculated on the policy network:



Supervised learning & Imitation Learning

Imitation Learning & Policy gradient

On-Policy vs Off-Policy

- REINFORCE is an **on-policy** algorithm because all the data used to improve the current policy are collected using the policy itself.

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ (run the policy)
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

- DQN is an **off-policy** algorithm because some of the data used to improve the current policy are collected using other policies.

Repeat:

- Take action a in current state s, observe r and s'; add experience tuple (s, a, s', r) to a buffer D; $s \leftarrow s'$
- Sample a minibatch $B = \{s_j, a_j, s'_j, r_j\}$ from D.
- Update the parameters

$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_j ([r(s_j, a_j) + \gamma \max_{a'_j} Q(s'_j, a'_j; \theta^-)] - Q(s_j, a_j; \theta))^2$$

- $\theta^- \leftarrow \theta$ in every C steps.

- Value-based RL:
  - Pros: Can be off-policy, can use experience reply, more sample efficient
  - Cons: Indirect, accuracy estimation of Q is difficulty, might lead to unstable policy.
- Policy-based RL:
  - Pros: Directly optimize policy, more stable.
  - Cons: It is on-policy, not sample efficient.
- Recent Trends: Combine the best of two worlds.

- Q-learning is off-policy even without experience replay because the agent does not necessarily take the action $a' = \arg\max_{a'} Q(s', a')$ in the next step
  - The action a' used for update is chosen using the current Q, but
  - The next action is chosen using the updated Q.

  - Initialize $Q(s, a)$ arbitrarily.
  - Repeat (for each episode)
    - Pick initial state s.
    - **Repeat**
      - Choose a for the state s (ϵ-greedy with $\arg\max_a Q(s, a)$)
      - Take action a, observe r and s'
      - Update:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

$$s \leftarrow s'$$

    - **until** s is terminal

$Q(s,a) \rightarrow Q_{K+1}$; $\max_{a'} Q(s',a') \rightarrow Q_K$

* Action used to estimate value of next state s'

$$\arg\max_{a'} Q_K(s', a')$$

* Action actually taken at next state

$$\arg\max_{a'} Q_{K+1}(s', a')$$

Sarsa Algo is on policy

- Initialize $Q(s, a)$ arbitrarily.
- Repeat (for each episode)
  - Pick initial state s.
  - Choose a for the state s (ϵ-greedy with $\arg\max_a Q(s, a)$)
  - **Repeat**
    - Take action a, observe r and s'
    - Choose a' for s' (ϵ-greedy with $\arg\max_a Q(s', a)$)
    - Update:

$$\begin{aligned} Q(s, a) &\leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)] \\ s &\leftarrow s', a \leftarrow a' \end{aligned}$$

  - **until** s is terminal

* Pick action for next state s' first

* It is used to estimate value of next state a'

* preparation: $V^*(S)$ vs $Q^*(S,a)$

Advantage function

Actor-Critic Algo vs REINFORCE

Estimation of the Advantage Function

Batch Actor-Critic algorithm

AKA: Advantage Actor-Critic Algo (A2C)

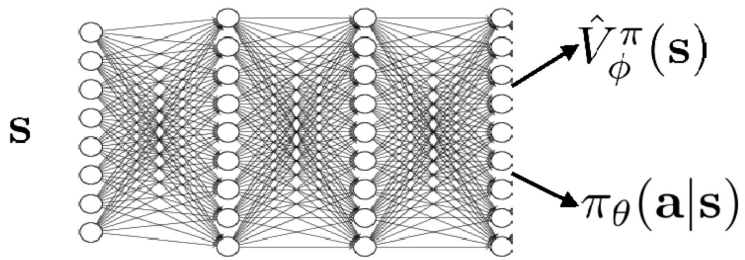
Repeat:

1. Sample experiences $\{(s_i, a_i, s_i', r_i)\}$ by following π_θ for multiple steps.
2. Update Critic parameters ϕ using L2 loss and training data

$$\{(s_i, r_i + \gamma \hat{V}^\pi_\phi(s_i'))\}$$

3. $\hat{A}^\pi(s_i, a_i) \leftarrow r_i + \gamma \hat{V}^\pi_\phi(s_i') - \hat{V}^\pi_\phi(s_i) \qquad \forall i$
4. Update actor parameters:

$$\theta \leftarrow \theta + \alpha \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}^\pi(s_i, a_i)$$



A2C in PyTorch : Colab

train_value_network

Repeat:

1. Sample experiences $\{(s_i, a_i, s_i', r_i)\}$ by following π_θ for multiple ste
2. Update Critic parameters ϕ using L2 loss and training data

$$\{(s_i, r_i + \gamma \hat{V}_\phi^\pi(s_i'))\}$$

3. $\hat{A}^\pi(s_i, a_i) \leftarrow r_i + \gamma \hat{V}_\phi^\pi(s_i') - \hat{V}_\phi^\pi(s_i) \qquad \forall i$
4. Update actor parameters:

$$\theta \leftarrow \theta + \alpha \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}^\pi(s_i, a_i)$$

```
# Enter no-gradient mode
with torch.no_grad():
    # Compute the next state values using the target network
    next_state_value = agent.target_network(next_states)
    # Zero out the next state values for the terminal states
    next_state_value[terminated] = 0
    # Compute the regression targets
    regression_targets = rewards + agent.gamma * next_state_value

# Compute the value predictions
value_predictions = agent.value_network(states)
# Compute the loss
loss = F.mse_loss(value_predictions, regression_targets)
# Zero the gradients
agent.value_optimizer.zero_grad()
# Compute the gradients
loss.backward()
# Update the weights
agent.value_optimizer.step()
```

A2C in PyTorch : Colab

+ train_policy_network

Repeat:

1. Sample experiences $\{(s_i, a_i, s'_i, r_i)\}$ by following π_θ for multiple steps.
2. Update Critic parameters ϕ using L2 loss and training data

$$\{(s_i, r_i + \gamma \hat{V}^\pi_\phi(s'_i))\}$$

3. $\hat{A}^\pi(s_i, a_i) \leftarrow r_i + \gamma \hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i) \qquad \forall i$
4. Update actor parameters:

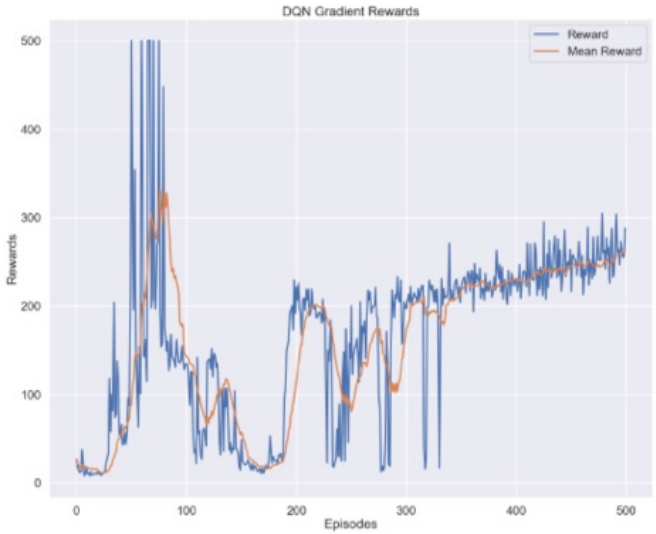
$$\theta \leftarrow \theta + \alpha \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}^\pi(s_i, a_i)$$

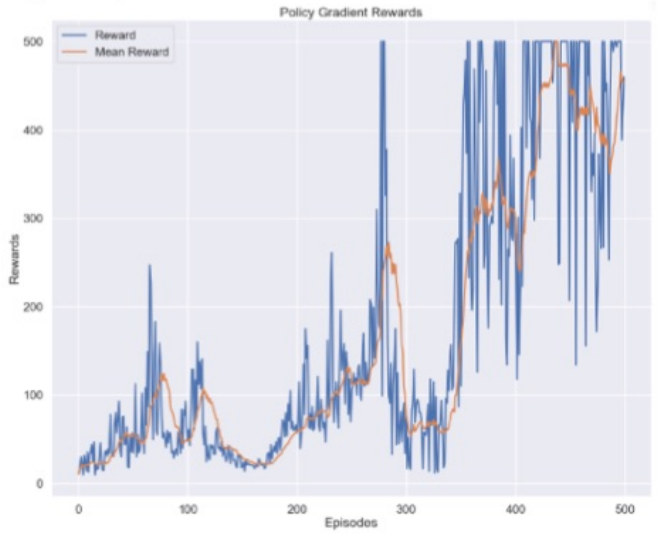
```
        with torch.no_grad():
            # Compute the next state values using the target network
            next_state_value = agent.target_network(next_state)
            # Zero out the next state values for the terminal states
            next_state_value[terminated] = 0
            # Compute the advantage
            advantage = reward + agent.gamma * next_state_value - agent.value_network(state)

        # Compute the log probability of the action
        log_prob = torch.log(agent.policy_network(state))[action]
        # Add advantage times log probability to the policy objective
        J += advantage * log_prob

    # Divide J by the number of datapoints in the batch
    J = J / len(batch)

    # Zero the gradients
    agent.policy_optimizer.zero_grad()
    # Compute the gradients
    J.backward()
    # Take a step with the optimiser
    agent.policy_optimizer.step()
```

- Value-based RL:
  - Pros: Can be off-policy, can use experience reply, more sample efficient
  - Cons: Indirect, accuracy estimation of Q is difficulty, might lead to unstable policy.
- Policy-based RL:
  - Pros: Directly optimize policy, more stable.
  - Cons: It is on-policy, not sample efficient.
- Recent Trends: Combine the best of two worlds.

Comparison of DQN, Policy Gradient and A2C on Cartpole (https://github.com/jordanlei/deep-reinforcement-learning-cartpole)

unstable policy





Not Sample efficient