UNIVERSITY OF BIRMINGHAM
COLLEGE OF ENGINEERING AND PHYSICAL SCIENCES

Final Project Presentation

Zhangda Xu 2088192
Supervisor: Mohan Sridharan

Contents

Introduction

- Goal: agent completes a tabletop task
- Method: Reinforcement learning

- Project rationale:
  1. define a problem
  2. formulate it
  3. solve it

Task of curling

- Curling:
  - Players slide stones on a sheet of ice toward a target area which is segmented into four concentric circles.
- Gameplay and RL
  - simulated environment
  - dedicated return signal
- Simplified task for agent:
  - Icy plane
  - one throw without sweeping the rock
  - Single round play
  - Arbitrary location of stones
  - Best score for all scenes

Fig 1. Curling stone

Goal

- Build tabletop environment and agent.
- Develop control methods of agent.
- Train agent in the face of uncertainty.
- Find policy for highest score in curling.

Reinforcement Learning

- Reward: a scalar feedback R_t

- At each step t the agent:
  - Executes action A_t
  - Observes environment O_t
  - Receives reward R_t

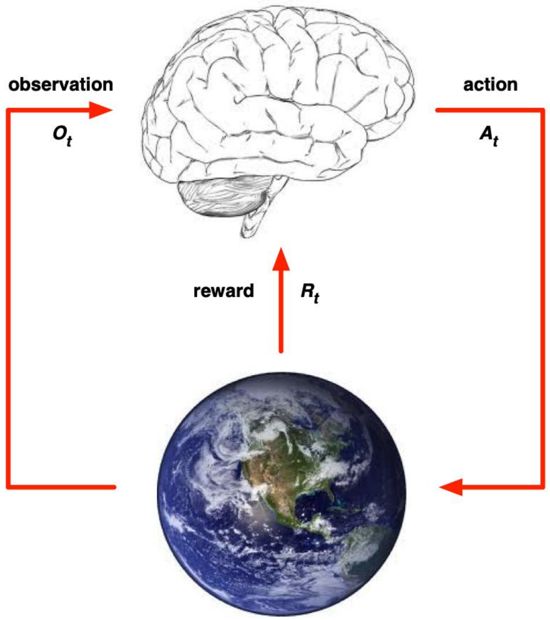

Fig 2. Environment and agent

Advantages:

- RL can deal with sequential data (non i.i.d.).
- RL has no supervisor but a returned reward.
- RL incorporates Interactions between agent and environment.

Markov Decision Process

- MDPs formally describe an environment for RL.

- A state S_t is **Markov** if and only if:

$$\mathbb{P}[S_{t+1} | S_t] = \mathbb{P}[S_{t+1} | S_1, \ldots, S_t]$$

- A MDP is an environment in which all states are **Markov**.
  - **A**: a finite action space to choose from.
  - **P**: a state transition probability matrix:

$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

  - **R**: a reward function for each state-action pair:

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

Tabular Representation

- **Policy**: maps states to a probability distribution over actions.

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]$$

- For an episodic MDP, the **state value function** is the expected return starting from state s, following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$$

- where **return** G_t is the total discounted reward from step t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \gamma \in [0,1]$$

- The **action-value function** is the expected return starting from state s, taking action a, following policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

Generalised Policy Iteration

- Bellman expectation equation with one-step lookahead DP:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s]$$

$$q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

- Bellman optimality equation:

$$v_*(s) = \max_a q_*(s,a)$$

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

- Finding the best action-value and the best policy.

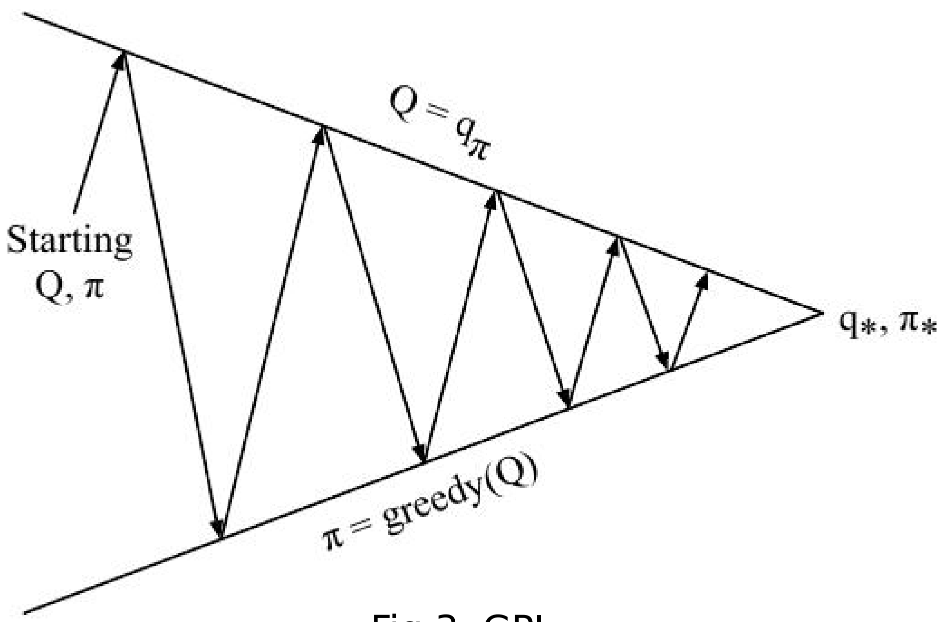

Fig 3. GPI

Task & Goal II

- Given the environment and agent:
- Accumulate reward in sequential states.
- Find the maximum total return G_t.

Temporal-difference Learning

- Goal: learn policy value function from experience under policy π.

- Update the state value online towards estimated return:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

- **Bootstrapping**: learn from incomplete episodes.
- **Sampling**: update with only one sample.

- TD(0) converges to MLE solution.

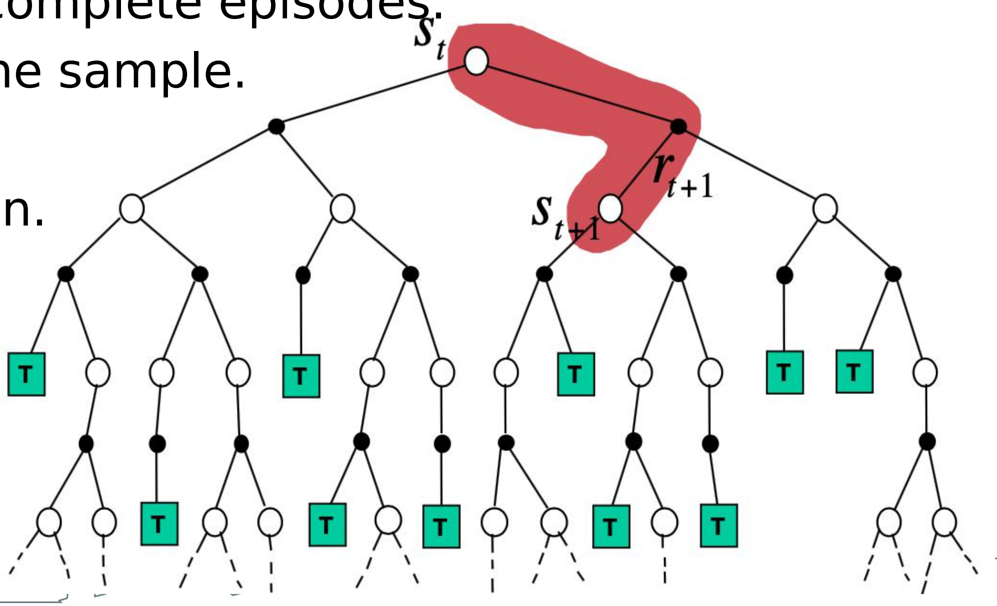

Fig 4. TD backup

Policy Gradient Methods

- **Approximation**: use parameters θ to generalise state-action pairs.
- Policy gradient algorithms search for a local maximum in any policy objective function by **ascending** the gradient of the policy *w.r.t.* parameters.
- For any **differentiable** policy, any of the policy objective functions, the policy gradient to full MDP is given as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)\right]$$

- Update parameters online using SGD.

Actor–critic

- Learning **off-policy**
- **Critic**: Updates action-value function parameters *w*.
- **Actor**: Updates policy parameters , in direction suggested by critic.

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\; Q_w(s,a)\right]$$
$$\Delta_\theta = \alpha \nabla_\theta \log \pi_\theta(s,a)\; Q_w(s,a)$$

- We need a policy gradient method with **lower variance**.
  - Using estimation instead of returned action value:

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$$

  - Using advantage function instead of action-value:

$$A(s,a) = Q_w(s,a) - V_v(s)$$

Task & Goal III

- Given the state and action:
- Approximate the reward for each action.
- Optimise the policy on curling gameplay.

Multi-layer Perceptron

- Feedforward neural network as universal approximator (Cybenko, 1989)

- Multi-layer perceptron (**MLP**) with at least one hidden layers:

$$\boldsymbol{H} = \phi\left(\boldsymbol{X}\boldsymbol{W}_h + \boldsymbol{b}_h\right)$$
$$\boldsymbol{O} = \boldsymbol{H}\boldsymbol{W}_o + \boldsymbol{b}_o$$

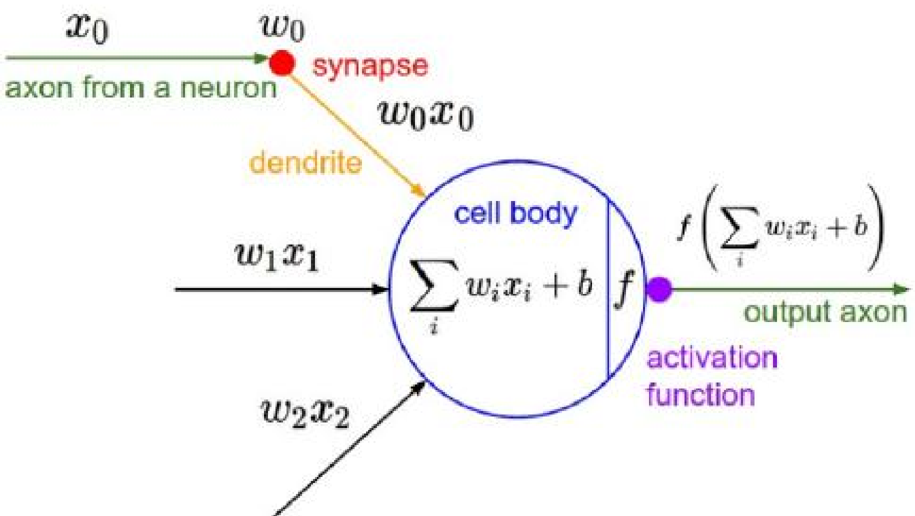

Fig 5. Multi-layer Perceptron

Deep learning applications:
- Computer vision
- Natural language processing
- Prediction
- Synthetic generator

Convolutional Filter

- **ConvNet** architecture:
  - Convolutional 2D
  - ReLU activation
  - Fully-connected
- Frame differences:
  - Episodic control task
  - Inference for velocity
- **End-to-end RL:**
  - Pixel input → ConvNet → Learning output

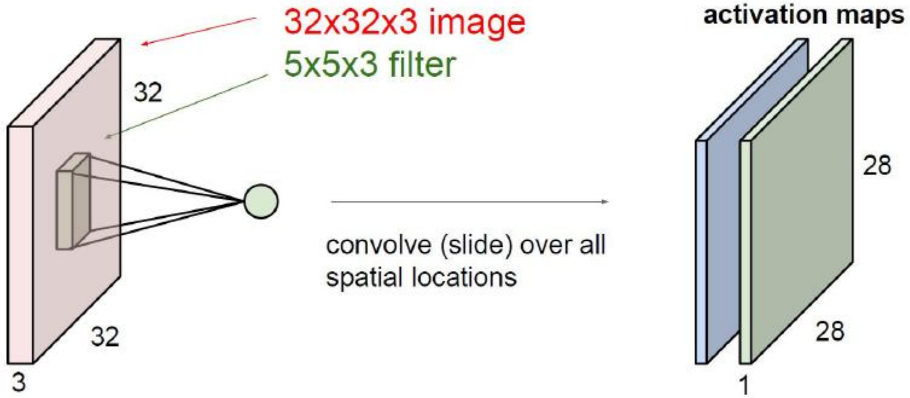

Fig 6. Conv layer

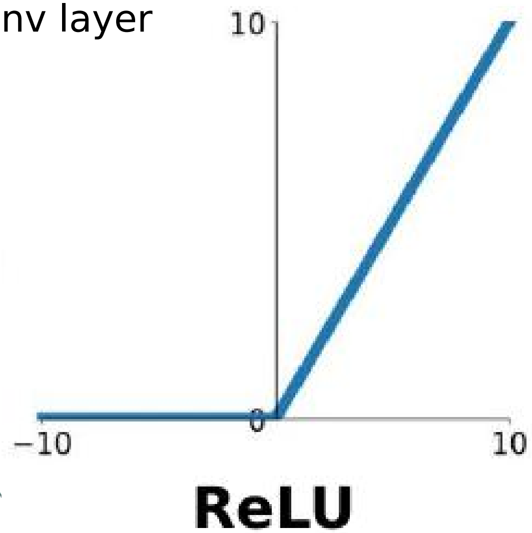

Fig 7. ReLU activation

Deep Reinforcement learning

- Standard policy update step in Q-learning:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(Y_t^{\mathrm{Q}} - Q(S_t, A_t; \boldsymbol{\theta}_t))\nabla_{\boldsymbol{\theta}_t} Q(S_t, A_t; \boldsymbol{\theta}_t)$$

- Q-learning (Watkins, 1990) uses the target:

$$Y_t^{\mathrm{Q}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t)$$

- TD-Gammon (Tesauro 1995)
- Deep Q Network (Mnih 2015):
  - Atari environment
  - For a given state *s*
  - Outputs a vector of action values *Q(s,·; θ).*
  - Read input directly from the screen frames.
  - Minimising the loss between action-value and target.

Deep Q-learning

- **Target network**: parameterised by θ^-
  - parameters copied every τ steps from the online network
  - kept fixed for other steps
  - update target:

$$Y_t^{\mathrm{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-)$$

- **Memory buffer**:
  - store samples of experiences
  - use random mini batch of experiences for update
  - break correlation between sequential data
    - better convergence

Task & Goal IV

- Observe the state from pixel input.
- Save memory after each throw.
- Update target and policy after several throws.

Deep Deterministic Policy Gradient

- Deterministic action function
  - unstable stochastic policy decision
  - slow optimisation for a_t
- Randomised exploration noise
  - continuous exploration
- Batch normalisation
  - minimise covariance shift
  - whitening features
- Action repeat
  - 3x simulated timesteps to infer velocity
- Soft update
  - slowly update the target parameters online

DDPG

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Twin-delayed DDPG

- Overestimate bias in actor-critic

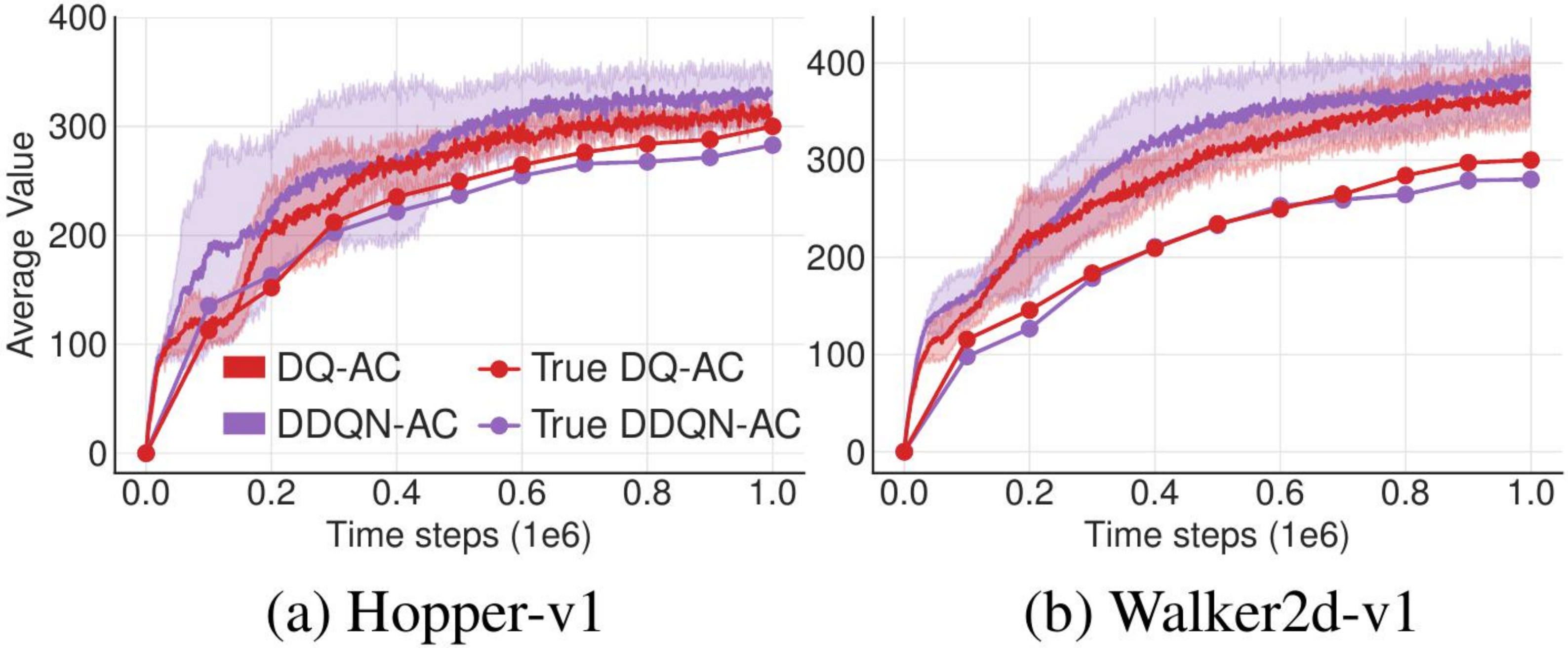

Fig 8. TD3 vs DDPG

TD3

- Fight against high variance value estimate:
  - **Double Q-learning**: value estimation disentanglement

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi}(s'))$$

  - **Clipped** Double Q-learning:

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_{\phi_1}(s'))$$

  - Delayed policy update:
    - Policy network should be updated at a **lower frequency** than the value network, to first minimise error before introducing a policy update.
  - Target Policy Smoothing Regularization:

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon),$$
$$\epsilon \sim \mathrm{clip}(\mathcal{N}(0, \sigma), -c, c),$$

Implementation

- |— Pybullet Client (no GUI)
  - |— puck.urdf - self-control
  - |— table.urdf - surface
  - |— stone.urdf - inertia
- |— gym-curling
  - |— curling_env.py
    - |— __init__()
    - |— step()
    - |— reset()
    - |— render()
- |— Algorithm
  - |— FIFO buffer
  - |— ddpg.py
  - |— td3.py
- |— PyTorch
  - |— Critic
  - |— Actor
  - |— Target critic
  - |— Target actor

PyBullet Gym Environment

- urdf models

```xml
<?xml version="1.0" ?>
<robot name="puck.urdf">
  <link name="baseLink">
    <contact>
      <lateral_friction value="0.0"/>
      <rolling_friction value="0.0"/>
      <stiffness value="300.0"/>
      <damping value="10.0"/>
    </contact>
    <inertial>
      <origin rpy="0 0 0" xyz="0 0 0"/>
       <mass value="1.0"/>
       <inertia ixx="1" ixy="0" ixz="0" iyy="1" iyz="0" izz="1"/>
    </inertial>
    <visual>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
                <mesh filename="cube.obj" scale="1 1 1"/>
      </geometry>
       <material name="white">
        <color rgba="1 1 1 1"/>
      </material>
    </visual>
    <collision>
      <origin rpy="0 0 0" xyz="0 0 0"/>
      <geometry>
        <box size="1 1 1"/>
      </geometry>
    </collision>
  </link>
</robot>
```

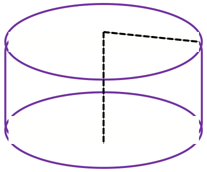

Fig 9. Stone/ Puck

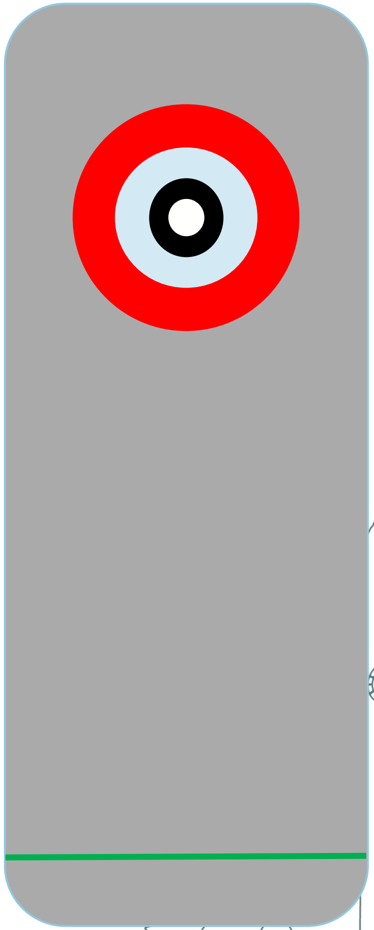

Fig 10. Tabletop

Hyperparameters

Action space	*(-1,1) * 5*
Observation space	(-1,1)^3
Reward definition	*R = [0, 1, 2, 3, 4]*
Viewpoint	0.7m upon centre
Action noise span	0.1
Memory buffer size	10^6
Epochs	100
Episode	1000

Training

1. Randomly initiate environment and target stones.
2. Place the ‘puck’ on the line.
3. Impose random starting velocity.
4. Render final environment state after collison stopped.
5. Store the state in memory buffer.
6. Update internal policy.

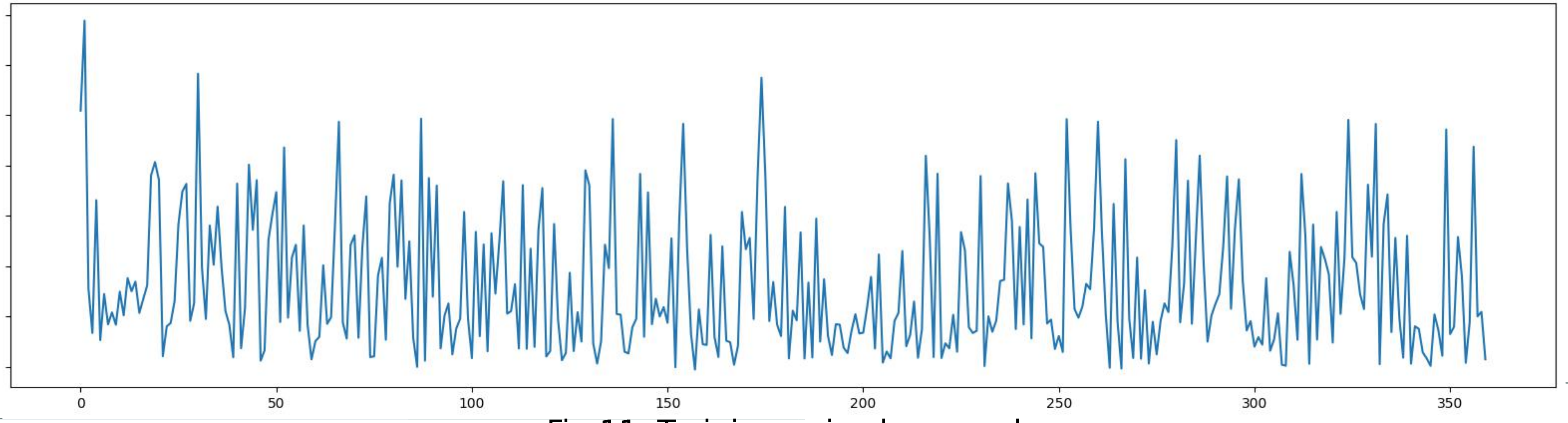

Fig 11. Training episode reward

Preliminary testing

Environment	TD3	DDPG	SAC	DQN (disc)
HalfCheetah	9636.95 ± 859.065	3305.60	2347.19	-15.57
Walker2d	4682.82 ± 539.64	1843.85	1283.67	2321.47
curling	9.2144 ± 0.5132	8.7313	6.5535	N/A

Future Experiment

- LSTM variant comparison
- Complete performance evaluation
- Refined gym environment GUI

- Other modifications:
  - intrisic exploration
  - prioritised experience replay
  - evolution strategies