10-703 Recitation 3

MCTS, TD Learning, Deep Q Learning, REINFORCE, Actor Critic
+ Homework 2 stuff

Athiya Deviyani

Monte Carlo

Monte Carlo (MC): Approach

Collect samples from your environment (state, action, reward trajectories).

$$\pi: S_0, A_0, R_1, S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T$$

Rather than storing all the rewards, we can do incremental update which uses visit count and the previous value function.

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

Where the return G is the sum of the discounted rewards

$$G_t = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

At the end of the trajectory you update each of the states that you've encountered with the average return.

$$V(S_t) \leftarrow average(Returns(S_t))$$

Monte Carlo (MC): Pseudocode

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-$

$G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T$

$G \leftarrow \gamma G + R_{t+1}$

Aggregate backwards

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow$ average$(Returns(S_t))$

You can do incremental update where you do one over n times the difference rather than holding all the returns in a list, use that as update rule.

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

How to generate policy from this? You will need Q-values by calculating the returns, and just do argmax of the Q

Monte Carlo vs. Dynamic Programming (Value/Policy Iteration)

Dynamic Programming (Value/Policy Iteration)	**Monte Carlo Learning**
Iterate through all of the states and update all of the states, which requires full knowledge of the reward function (the transition function and reward function) $\Sigma_{s',r}\, p(s', r \mid s, a)\left[r + \gamma V(s')\right]$	Do a single trajectory, update all the states within that trajectory. It assumes that all the trajectories are episodic (terminates at time step T) $\pi: S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
Biased estimate: one-step bootstrap	Unbiased estimate: average over the returns, but higher variance

Monte Carlo vs. Dynamic Programming Backup Diagrams

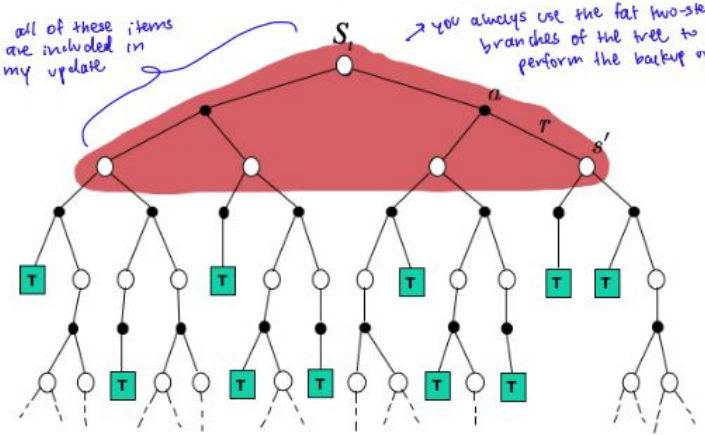
Dynamic Programming

$$V(S_t) \leftarrow E_\pi\left[R_{t+1} + \gamma V(S_{t+1})\right] = \sum_a \pi(a|S_t) \sum_{s',r} p(s', r|S_t, a)[r + \gamma V(s')]$$

you have true dynamic probabilities

all of these items are included in my update

you always use the fat two-step branches of the tree to perform the backup update



Only goes down one layer and uses the value function from the next state to compute the value function of the current state, and it does this across the future states.

Monte Carlo

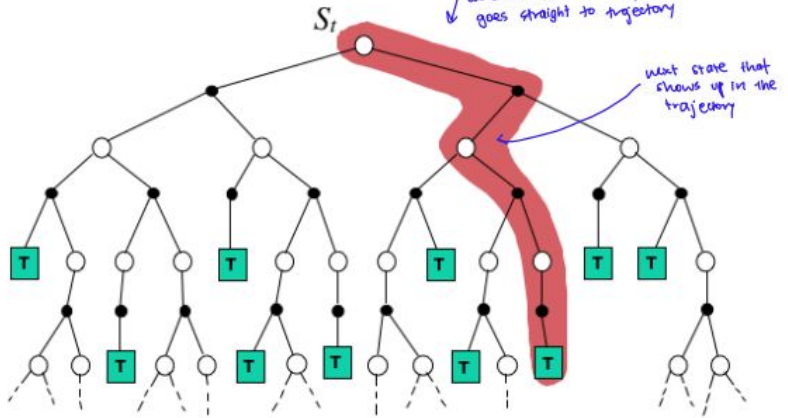
$$V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t))$$

sums the rewards from S_t onwards

root of tree

does not use 'fat' update goes straight to trajectory

next state that shows up in the trajectory



Goes all the way down the tree until the terminal state and only update states along this trajectory (does not consider multiple different possibilities)

Temporal Difference and Q-Learning

Temporal Difference Learning (TD Learning)

Monte-Carlo requires episodic trajectories (termination), otherwise it won't work! Additionally, if the trajectory is very long, using MC will require a lot of memory and computational resources.

Replacement: use value function to approximate the sum of the future discounted return

New update rule:

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

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T$$

target: an estimate of the return

+ Can learn before reaching a terminal state
+ Much more memory and computation-efficient than MC
- Using value in the target introduces bias

If alpha is very small, it is actually equivalent to the 1/N(S_t)

TD Learning: Approach

- Monte Carlo learns at the end of each episode, TD Learning learns at each step, i.e. update $V(S_t)$ at each step
- Since we haven't completed an episode, we don't have the expected return G_t, so we need to estimate it
  - This estimation is called **bootstrapping**, because TD bases its update in part on an existing estimate $V(S_{t+1})$ and not a complete sample G_t

TD Learning: Pseudocode

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\mathit{terminal}) = 0$

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        $A \leftarrow$ action given by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until S is terminal

TD Learning: N-step returns

TD(0)

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+2}|S_t = s\right] \\
&= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s\right]
\end{aligned}
$$

MC estimate

Approximate with v

N-step returns

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s\right] \\
&= \mathbb{E}_\pi\left[\sum_{i=1}^{N}\gamma^{i-1} R_{t+i} + \gamma^N\sum_{k=0}^{\infty}\gamma^k R_{t+k+N+1}|S_t = s\right] \\
&= \mathbb{E}_\pi\left[\sum_{i=1}^{N}\gamma^{i-1} R_{t+i} + \gamma^N v_\pi(S_{t+N})|S_t = s\right]
\end{aligned}
$$

N-step returns

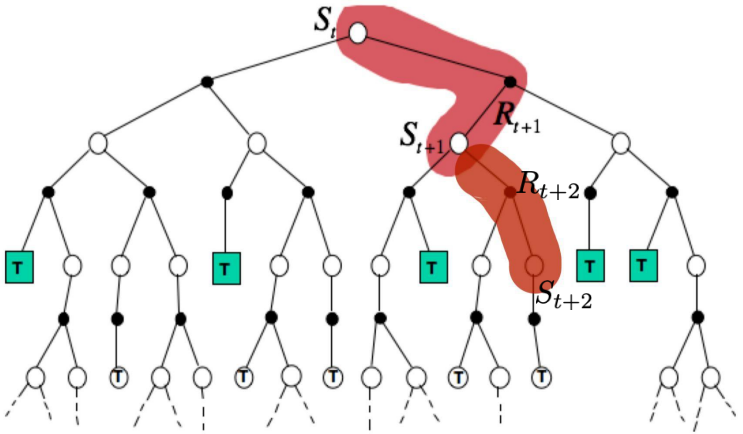
Less reliance on v

11

Further develop TD to have N-step returns. Use exact return for N steps, and starting from step N+1, we use the value function to estimate the future returns. N-step TD learning will have less reliance on the value function. Unlike the original TD(0), the N-step TD will be much more stable.

TD Learning: N-step returns example with N=2

$$V(S_t) \leftarrow V(S_t) + \alpha \left(R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) - V(S_t) \right)$$



Q-learning: Off-policy TD Learning

Q-learning is an off-policy value-based method that uses a TD approach to train its action-value (Q) function.

1-step Q-learning update

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

- Key benefit: off-policy (the policy you are using to select the action is different from the policy you are 'learning'; on-policy methods attempt to evaluate or improve the policy that is used to make decisions)
- Only require state, action, reward, and next state drawn from the MDP
- Doesn't depend on the policy anywhere!
- Is foundation for many sample-efficient RL methods

Q-learning: Pseudocode

Algorithm 14: Sarsamax (Q-Learning)

Input: policy π, positive integer *num_episodes*, small positive fraction α, GLIE $\{\epsilon_i\}$
Output: value function Q ($\approx q_\pi$ if *num_episodes* is large enough)
Initialize Q arbitrarily (e.g., $Q(s,a) = 0$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, and $Q(\textit{terminal-state}, \cdot) = 0$)
for $i \leftarrow 1$ ***to*** *num_episodes* **do** Step 1
  $\epsilon \leftarrow \epsilon_i$
  Observe S_0
  $t \leftarrow 0$
  repeat
    Choose action A_t using policy derived from Q (e.g., ϵ-greedy) Step 2
    Take action A_t and observe R_{t+1}, S_{t+1} Step 3
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t))$ Step 4
    $t \leftarrow t + 1$
  until S_t *is terminal*;
end
return Q

Q-learning: Approach

1. Initialize the Q-table for each state-action pair arbitrarily (e.g. set all of the values to 0)
2. Choose an action using the epsilon-greedy strategy
    a. As training goes on, our estimates become better, so it is useful to progressively reduce the epsilon value (so we exploit more than we explore)
3. Perform action A_t, get reward R_{t+1} and next state S_{t+1}
4. Update $Q(S_t, A_t)$
    a. We update our policy or value function after one step of the interaction
    b. Produce the TD target by using the immediate reward R_{t+1} plus the discounted value of the next state
        i. This is computed by finding the action that maximizes the current Q-function at the next state (we use a greedy policy to select the next best action)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

New Q-value estimation

Former Q-value estimation

Learning Rate

Immediate Reward

Discounted Estimate optimal Q-value of next state

Former Q-value estimation

TD Target

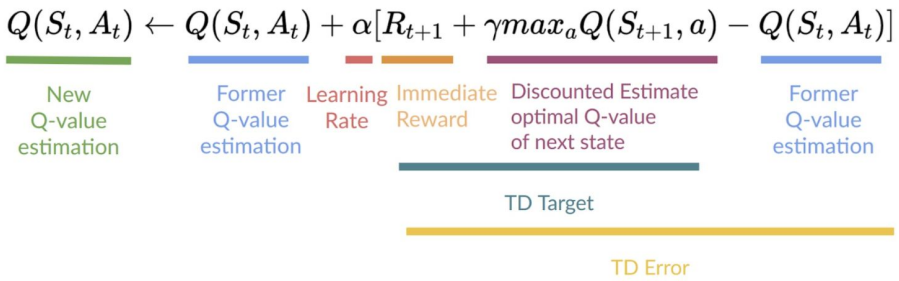
TD Error

Deep Q learning

Source: https://huggingface.co/learn/deep-rl-course/unit3/deep-q-algorithm

Deep Q-learning: Overview

- In Q-learning, we directly update the Q-value of a state-action pair directly. If the state space and action space are too large, we can use a function approximator (neural network) to approximate the Q-values!



- In deep Q-learning, we create a loss function that compares our Q-value prediction and the Q-target and uses gradient descent to update the weights of our Deep Q-Network to approximate our Q-values better

$$L = \left(\text{sg}(R_{t+1} + \gamma \max_{A_{t+1}} q(S_{t+1}, A_{t+1}, w)) - q(S_t, A_t, w) \right)^2$$

Target value

Prediction

Deep Q-learning: Algorithm

Algorithm 4 DQN

procedure DQN

Initialize network Q_ω *and* Q_{target} *as a clone of* Q_ω

Initialize replay buffer R *and burn in with trajectories followed by random policy*

Initialize $c = 0$

repeat for E **training episodes:**

Initialize S_0

for $t = 0, 1, \ldots, T-1$:

$$a_t = \begin{cases} \arg\max_a Q_\omega(s_t, a) & \text{with probability } 1-\epsilon \\ \text{Random action} & \text{otherwise} \end{cases}$$

Take a_t *and observe* r_t, s_{t+1}

Store (s_t, a_t, r_t, s_{t+1}) *in* R

Sample minibatch of (s_i, a_i, r_i, s_{i+1}) *with size* N *from* R

$$y_i = \begin{cases} r_i & s_{i+1} \text{ is terminal} \\ r_i + \gamma \max_a Q_{\text{target}}(s_{i+1}, a) & \text{otherwise} \end{cases}$$

$L(\omega) = \frac{1}{N} \sum_{i=0}^{N-1} (y_i - Q_\omega(s_i, a_i))^2$

Update Q_ω *using* `Adam` $(\nabla_\omega L(\omega))$

$c = c + 1$

Replace Q_{target} *with current* Q_ω *if* $c \ \% \ 50 = 0$

end procedure

Sampling: we perform actions and store the observed experience tuples in a replay memory

Training: select a small batch of tuples randomly and learn from this batch using a gradient descent update step

Deep Q-learning: Limitations and Solutions

Because deep Q-learning combines a non-linear Q-value function (Neural network) with bootstrapping (when we update targets with existing estimates and not an actual complete return), it might suffer from instability.

To help us stabilize the training, we implement three different solutions:

1. Experience Replay to make more efficient use of experiences.
2. Fixed Q-Target to stabilize the training.
3. Double Deep Q-Learning, to handle the problem of the overestimation of Q-values.

Deep Q-learning: Experience Replay

Algorithm 4 DQN

procedure DQN
  Initialize network Q_ω *and* Q_{target} *as a clone of* Q_ω
  Initialize replay buffer R *and burn in with trajectories followed by random policy*
  Initialize $c = 0$
  repeat for E **training episodes:**
    Initialize S_0
    for $t = 0, 1, \ldots, T-1$:
      $$a_t = \begin{cases} \arg\max_a Q_\omega(s_t, a) & \text{with probability } 1-\epsilon \\ \text{Random action} & \text{otherwise} \end{cases}$$
      Take a_t *and observe* r_t, s_{t+1}
      Store (s_t, a_t, r_t, s_{t+1}) *in* R
      Sample minibatch of (s_i, a_i, r_i, s_{i+1}) *with size* N *from* R
      $$y_i = \begin{cases} r_i & s_{i+1} \text{ is terminal} \\ r_i + \gamma \max_a Q_{\text{target}}(s_{i+1}, a) & \text{otherwise} \end{cases}$$
      $L(\omega) = \frac{1}{N} \sum_{i=0}^{N-1} (y_i - Q_\omega(s_i, a_i))^2$
      Update Q_ω *using* `Adam` $(\nabla_\omega L(\omega))$
      $c = c + 1$
      Replace Q_{target} *with current* Q_ω *if* $c \ \% \ 50 = 0$
end procedure

Uses the experiences of the training more efficiently (we can use a replay buffer that saves experience samples that we can reuse during sampling)

- Agent can learn from the same experience multiple times!

Avoid forgetting previous experiences and reduce the correlation between experiences

- if we give sequential samples of experiences to our neural network is that it tends to forget the previous experiences as it gets new experiences

By randomly sampling experiences, we remove correlation in the observation sequences to avoid actin values from oscillating or diverging catastrophically.

Deep Q-learning: Target Network

$$L = \left(\text{sg}(R_{t+1} + \gamma \max_{A_{t+1}} q(S_{t+1}, A_{t+1}, w)) - q(S_t, A_t, w) \right)^2$$

Target value

Prediction

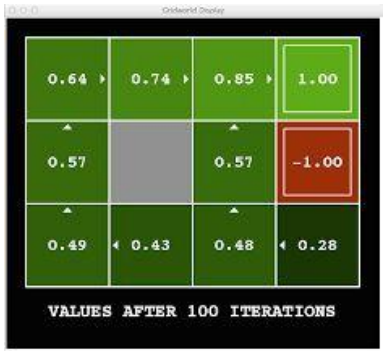
- **Problem**: nonstationary targets
  - Where the most instability comes from
  - Updating the network weights changes the target value, which requires more updates
  - Unintended generalization to other states S' can lead to error propagation
- **Solution**: calculate target values with a network that's updated every T gradient steps
  - For T steps, the target network is fixed, after that you update the target network once and continue to update your value function for another T steps, repeat the process
  - Network has more time to fit targets accurately before they change
  - Slows down training, but not too many alternatives (recently: functional regularization)

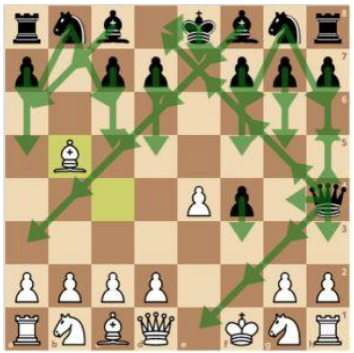
Monte-Carlo Tree Search

Monte-Carlo Tree Search: Background Motivation

Problem: Large State-Action Space

- Trying to estimate the value at every state (solving the full MDP) is often infeasible





- MC and TD still try to estimate Q/V value function for every state or state-action visited
  - Too much memory for tabular (e.g. for chess, this would be 10^48 states)
  - Neural Network may be undefined at unseen states, and ‘similar’ states may have completely different values and optimal paths (TD and MC rely on the fact that every state and every action has been visited)

Monte-Carlo Tree Search: Definitions

- Planning: any computational process that uses a model to create or improve a policy
  - Given a model environment, come up with the best policy
- Online planning: unroll the model of the environment forward in time to select the right action sequences to achieve your goal
  - On the fly, while you're playing the game you're in a particular state, you use the knowledge of the model of the environment model to unroll it forward and evaluate all possible alternatives
  - Limited by resources

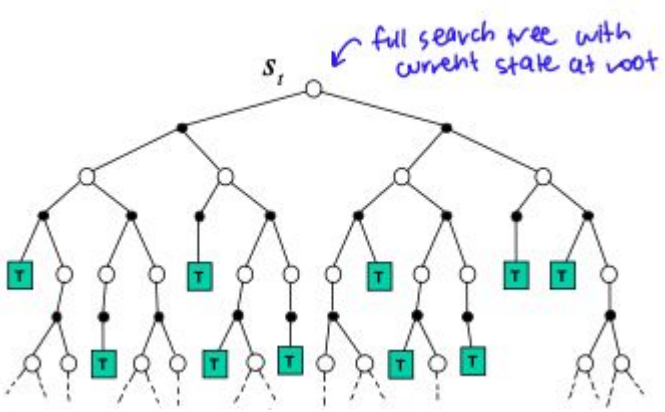
Monte-Carlo Tree Search: vs. Online Planning

Online planning

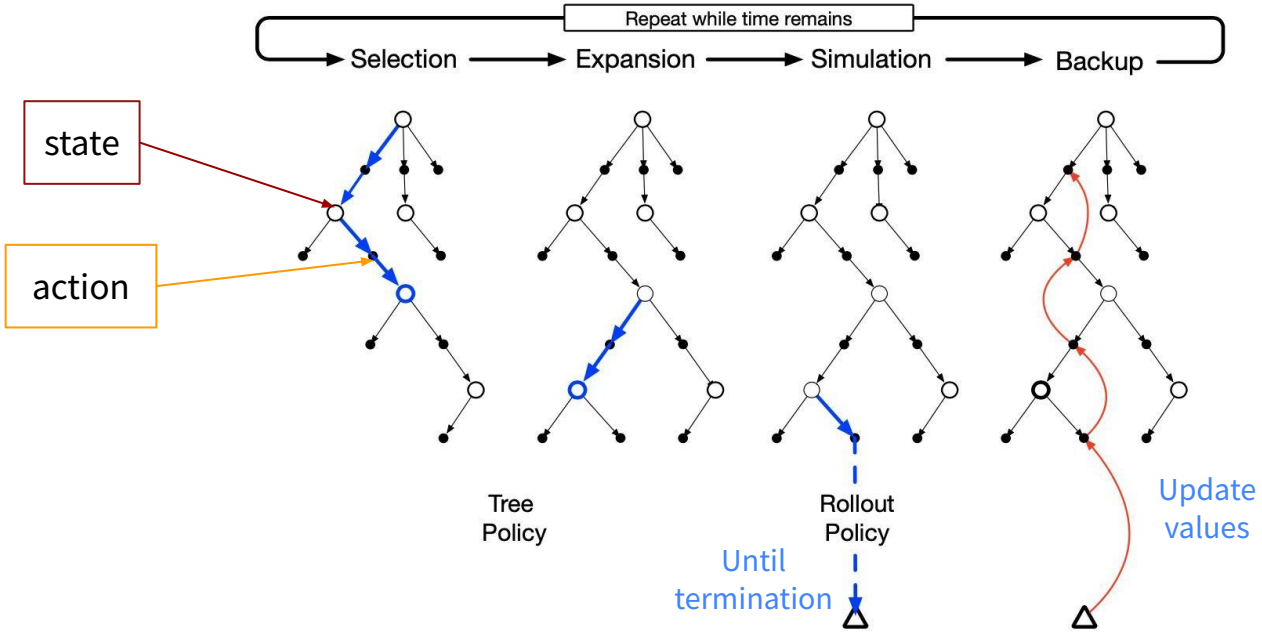
- Use internal model to simulate trajectories at current state, find the best one
- Problems: curse of dimensionality
  - Too many actions possible: large tree branching factor
  - Too many steps: large tree depth

MCTS

- Only estimate value function for relevant part of state space
- Consider only part of the full MDP at a given step



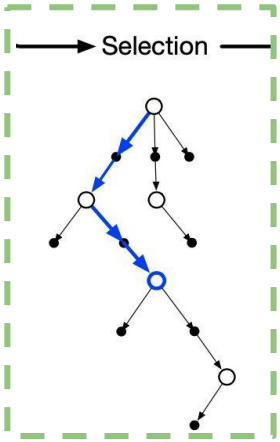
Monte-Carlo Tree Search: Overview



Tree: stores Q-values for only a subset of all state-actions (stores Q-values for those)
MC-method: require episode termination to update values

Monte-Carlo Tree Search: Selection

Inside the search tree



Given: current state of agent (root node), empty or existing tree with Q-values

Steps

"Children" here refers to actions, so in this step we look at possible actions from the current state

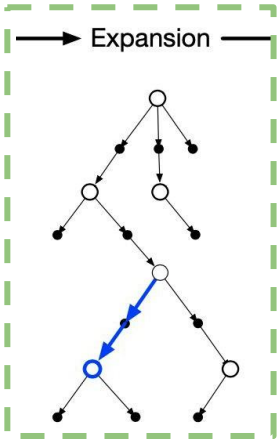
```
function MCTS_sample(node)
   if all children expanded:#selection
      next = UCB_sample(node)
      outcome = MCTS_sample(next)
```

Where UCB_sample is $A_t = \text{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right]$

Keep doing UCB repeatedly until you reach frontier of the tree (unexplored state). For every state we bookkeep the number of visits and wins.

Monte-Carlo Tree Search: Expansion

Inside the search tree



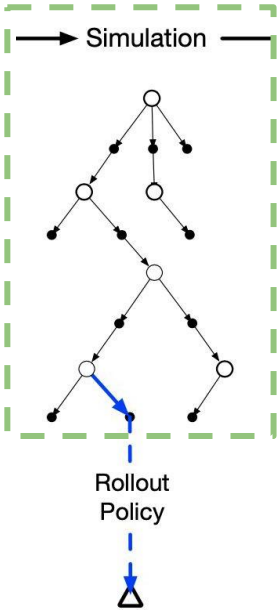
Given: a new state *s* not part of the tree (state that you have not seen before; unexpanded child)

Steps

- Based on some rule (e.g. state < max depth), possibly add this new state to the tree
- Take random action (since no Q-values are available), receive reward *r* if available
- Calculate return, G = Simulation(s, a)
- Store Q(s, a) = gamma*G + r
- Return gamma*G + r to propagate return to parent node

Monte-Carlo Tree Search: Simulation

Inside the search tree



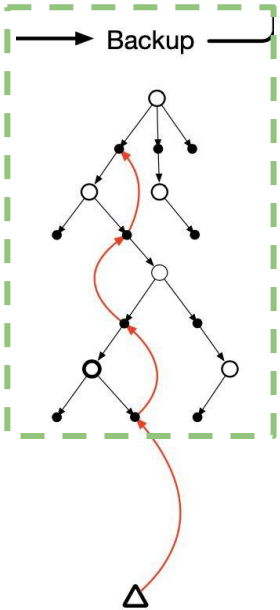
Given: a new state *s* not part of the tree

Steps:

- If state is terminal, return reward
- Else, use a very fast policy to determine action *a* to take
  - Most commonly used policy: random policy
- G = Simulation(s, a)
  - Main difference with previous stage: we do not store Q-value!
- Return gamma*G + r

Monte-Carlo Tree Search: Simulation

Inside the search tree



Propagate return from the recursive calls

Calculate the return at each state

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots + \gamma^{T-1} R_T$$

Update visitation count and value of each visited state

```
function update_value(node, outcome):
   #combine the new outcome with the average value
   node.value *= node.visits
   node.visits++
   node.value += outcome
   node.value /= node.visits
```

Monte-Carlo Tree Search: Summary

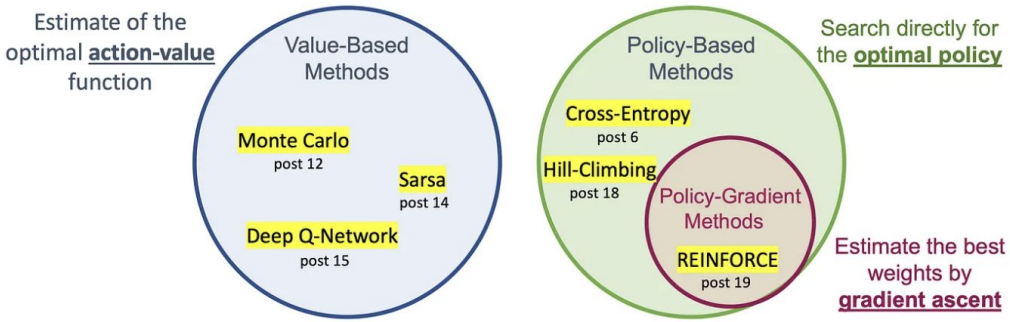
- For the current state of agent, repeatedly perform the previous steps until some stopping criteria is reached
  - Examples: time limit, Q-value convergence within some threshold
- Execute the best action (select action with the highest Q-value estimate)
- Reuse the subtree of the successor state and repeat

When to use MCTS over learning algorithms?

- More useful if you have limited amount of time
- Access to internal model (environment dynamics)
- Size or dynamic nature of the state-action space (in MCTS, the state action space size doesn't matter because it only explores the best actions)

REINFORCE

REINFORCE: Policy-based methods



- Value-based methods: learn a value function (an optimal value function leads to an optimal policy)
  - Goal: minimize the loss between the predicted and target value
  - Policy is implicit as it is generated directly from the value function (e.g. eps-greedy from Q-function)
  - Examples: Monte-Carlo, DQN, SARSA
- Policy-based methods: learn to approximate optimal policy directly (without learning a value function)
  - Parameterize the policy, e.g. using a neural network
  - Policy outputs a probability distribution over actions (stochastic policy)
  - Goal: maximize the performance of the parameterized policy using gradient ascent

REINFORCE: Policy Gradient algorithm

Goal: control the probability distribution of actions by tuning the policy such that good actions (that maximizes the return) are sampled more frequently in the future

Optimization method: let agent interact with the environment during an episode; if we win the episode, we want to increase P(a|s), and decrease if we lose

Training Loop:

Collect an **episode with the π** (policy).
Calculate the return (sum of rewards).

Update the weights of the π:

If **positive return** → **increase** the probability of each (state, action) pairs taken during the episode.

If **negative return** → **decrease** the probability of each (state, action) taken during the episode

REINFORCE: Algorithm

REINFORCE, or Monte-Carlo policy-gradient, uses an estimated return from an entire episode to update the policy parameter θ.

In a loop,

1. Use the policy π_θ to collect episode τ
2. Use the episode to estimate the gradient g = $\nabla \theta J(\theta)$

$$\nabla_\theta J(\theta) \approx \hat{g} = \sum_{t=0} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)$$

Estimation of the gradient (given we use only one trajectory to estimate the gradient)

Probability of the agent to select action at from state st given our policy

Cumulative return

Direction of the steepest increase of the (log) probability of selecting action at from state st

3. Update the weights of the policy: $\theta \leftarrow \theta + \alpha g$

REINFORCE: Algorithm

REINFORCE, or Monte-Carlo policy-gradient, uses an estimated return from an entire episode to update the policy parameter θ.

In a loop,

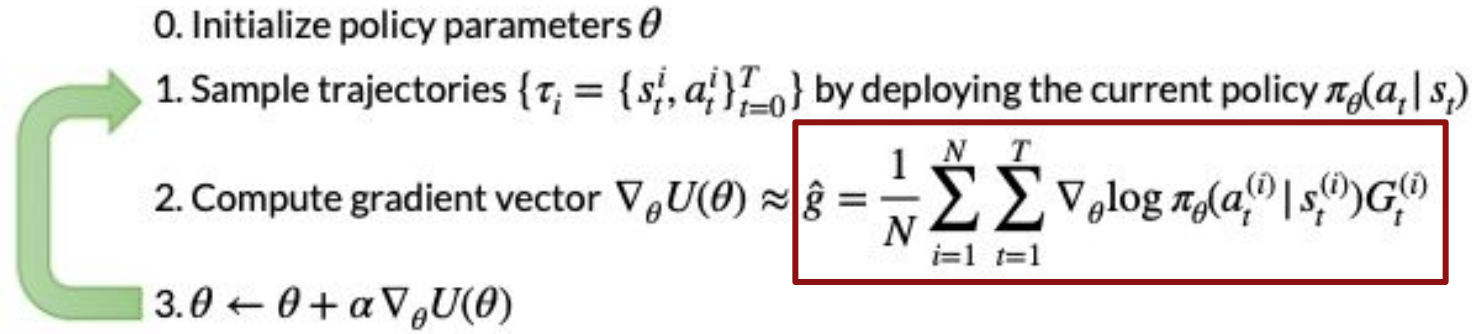
1. Use the policy π_θ to collect episode τ
2. Use **multiple episodes** to estimate the gradient g = $\nabla_\theta J(\theta)$

$$\nabla_\theta J(\theta) \approx \hat{g} = \frac{1}{m} \sum_{i=1}^{m} \sum_{t=0} \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) R(\tau^{(i)})$$

- Estimation of the gradient (given we use some trajectories to estimate the gradient)
- Scaling factor inversely proportional to the number of trajectories (m)
- Probability of the agent to select action at from state st given our policy in trajectory (i)
- Direction of the steepest increase of the (log) probability of selecting action at from state st
- Cumulative return of i-th trajectory

3. Update the weights of the policy: θ ← θ + αg

REINFORCE: The Variance Problem



The gradient estimator above is unbiased, i.e. with large N it will accurately approximate the true gradient.

Problem: usually, a very large N is required

How do we minimize the variance of our estimator?

$$\text{Var}(\hat{g}) = \text{tr}\left(\mathbb{E}\left[(\hat{g} - \mathbb{E}[\hat{g}])(\hat{g} - \mathbb{E}[\hat{g}])^T\right]\right) = \sum_{k=1}^{n}\mathbb{E}\left[\left(\hat{g}_k - \mathbb{E}[\hat{g}_k]\right)^2\right]$$

REINFORCE: Reducing Variance

- Policy gradient methods suffer from high variance caused by the empirical returns
- We can reduce the variance by subtracting a baseline from the returns in the policy gradient, as it will make smaller gradients (thus more stable updates!)
  - The baseline is a proxy for the expected actual return that does not introduce any bias to the policy gradient
  - Good example of a baseline is the value function: policy gradient - value function baseline = advantage

$$\begin{aligned}\nabla_\theta V(\theta) &= E[\sum_{t=0}^{T-1} \nabla_\theta \log_\theta(a_t, s_t)(G_t - b(s_t)] \\ &= E[\sum_{t=0}^{T-1} \nabla_\theta \log_\theta(a_t, s_t)(G_t - V(s_t; w)] \\ &= E[\sum_{t=0}^{T-1} \nabla_\theta \log_\theta(a_t, s_t) A_t]\end{aligned}$$

  - You can also use a parametrized model Q(s,a) to approximate the value in the advantage instead of using the empirical returns, and this method is called **Actor-Critic**

Source: https://medium.com/p/e95ace11c1c4

REINFORCE: Policy-based methods, pros and cons

Pros

- We can estimate the policy directly without storing additional data
- Policy-gradient methods can learn a stochastic policy
  - We don’t need to implement an exploration/exploitation trade-off by hand
- More effective in high-dimensional action spaces and continuous action spaces
- Better convergence properties

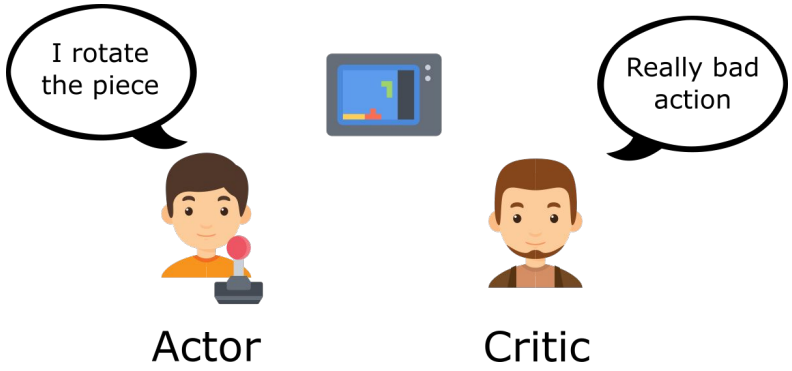
Cons

- Converges to a local maximum sometimes
- Slower, step-by-step: it can take longer to train (inefficient)
- Gradient estimate is very noisy: there is a possibility that the collected trajectory may not be representative of the policy
- High variance

Actor Critic

Source: https://huggingface.co/learn/deep-rl-course/unit6/advantage-actor-critic

Actor-Critic: Overview



In Actor-Critic methods, we learn two function approximations

- A policy that controls how our agent acts, or the **actor**
- A value function to assist the policy update by measuring how good the action taken is, or the **critic**

Actor-Critic: Algorithm

0. Initialize policy parameters θ and critic parameters ϕ.

1. Sample trajectories $\{\tau_i = \{s_t^i, a_t^i\}_{i=0}^T\}$ by deploying the current policy $\pi_\theta(a_t \mid s_t)$

2. Fit value function $V_\phi^\pi(s)$ by MC or TD estimation (update ϕ) → based on policy

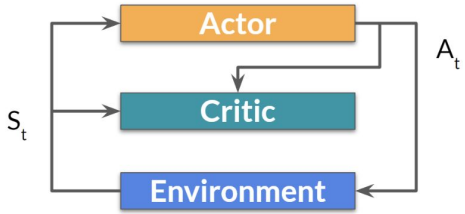
3. Compute action advantages $A^\pi(s_t^i, a_t^i) = G_t^{(i)} - V_\phi^\pi(s_t^i)$ — regression; use value functions

4. $\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i) A^\pi(s_t^i, a_t^i)$

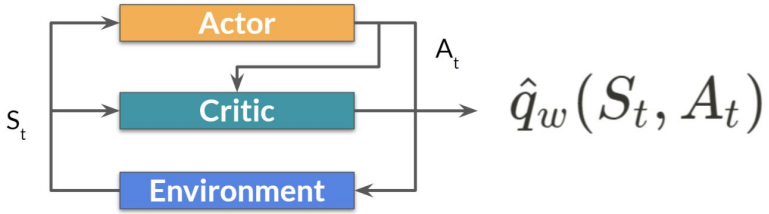
5. $\theta \leftarrow \theta + \alpha \nabla_\theta U(\theta)$ ← update the policy network

Actor-Critic: Algorithm

1. At each time step t, we get the current state S_t from the environment and pass it as input through our Actor and Critic model
2. Our policy takes the state and outputs action A_t

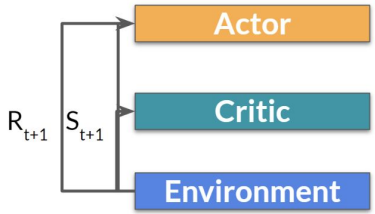


3. The Critic model takes that action also as input and computes the value of taking that action at that state using S_t and A_t (the Q-value)



Actor-Critic: Algorithm

4. The action A_t performed in the environment outputs a new state S_{t+1} and a reward R_{t+1}



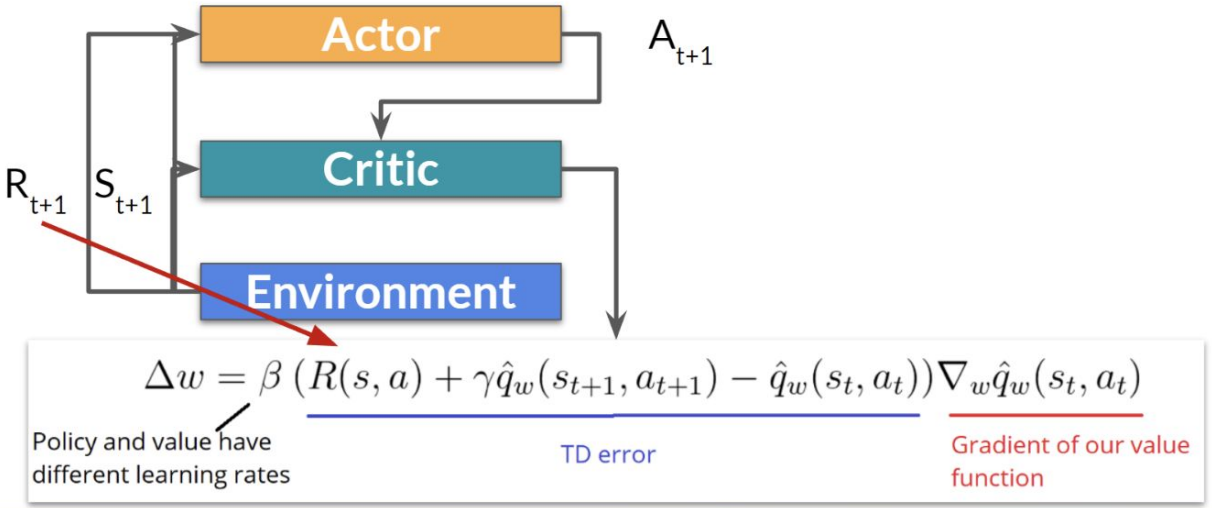
5. The Actor model updates its policy parameters using the Q-value

$$\Delta\theta = \alpha\nabla_\theta(log\pi_\theta(s,a))\hat{q}_w(s,a)$$

Change in policy parameters (weights) Action value estimate

Actor-Critic: Algorithm

6. Then, the Actor model produces the next action to take at A_t given the new state S_{t+1}
7. The Critic model then updates its parameters



Actor-Critic: Advantage Actor-Critic (A2C)

- When we use the Advantage function as the Critic instead of the Action value function, we can stabilize learning further
  - The Advantage function calculates the relative advantage of an action compared to the others possible at a state (*how is taking that action at a state better compared to the average value of the state?*)

$$A(s,a) = Q(s,a) - V(s)$$

q value for action a in state s

average value of that state

Actor-Critic: Advantage Actor-Critic (A2C)

$$A(s,a) = Q(s,a) - V(s)$$

- If A(s,a) > 0: our gradient is pushed in that direction
- If A(s,a) < 0: our gradient is pushed in the opposite direction

The problem with implementing this advantage function is that it requires two value functions - Q(s,a) and V(s). Fortunately, we can use the TD error as a good estimator of the advantage function.

$$A(s,a) = Q(s,a) - V(s)$$

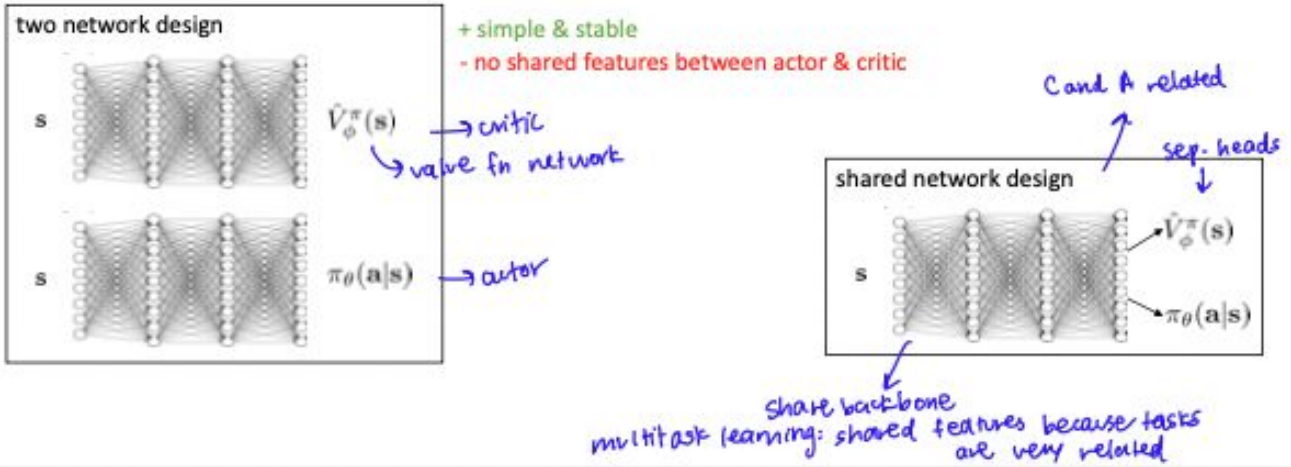
$$Q(s,a) = r + \gamma V(s')$$

$$A(s,a) = r + \gamma V(s') - V(s)$$

TD Error

Actor-Critic: Architecture

Architecture choices



Code walkthrough

Part 1: REINFORCE, REINFORCE+Baseline, A2C

```python
import sys
import argparse
import numpy as np

import torch

# set random seeds
torch.manual_seed(0)
np.random.seed(0)

class A2C(object):
    # Implementation of N-step Advantage Actor Critic.

    def __init__(self, actor, actor_lr, N, nA, critic, critic_lr, baseline=False, a2c=True):
        # Note: baseline is true if we use reinforce with baseline
        #       a2c is true if we use a2c else reinforce
        # TODO: Initializes A2C.
        self.type = None  # Pick one of: "A2C", "Baseline", "Reinforce"
        assert self.type is not None
        pass

    def evaluate_policy(self, env):
        # TODO: Compute Accumulative trajectory reward(set a trajectory length threshold if you want)
        pass

    def generate_episode(self, env, render=False):
        # Generates an episode by executing the current policy in the given env.
        # Returns:
        # - a list of states, indexed by time step
        # - a list of actions, indexed by time step
        # - a list of rewards, indexed by time step
        # TODO: Implement this method.
        pass

    def train(self, env, gamma=0.99, n=10):
        # Trains the model on a single episode using REINFORCE or A2C/A3C.
        # TODO: Implement this method. It may be helpful to call the class
        #       method generate_episode() to generate training data.
        pass
```

Init: initialize the type that you are going to use, the model(s) that you are going to use, Adam optimizer

Evaluate_policy: Run through the policy *once* to obtain the return from a single trajectory (where return is the sum of the rewards)

Generate_episode: Collect state, action, reward pairs (trajectories) by executing the current policy

Hint: you might also want to return the action probabilities here to avoid recalculation in the future

Train: Train the model(s), calculate the loss, backpropagate the loss, *zero the gradients*

Hint: when doing A2C, be careful about detaching the gradients of the actor/critic (when you are updating the loss of the actor, make sure to detach values coming from the critic used in the loss update)

Tip: implement REINFORCE, REINFORCE+Baseline, and A2C sequentially as they build off of each other

Part 2: Deep Q-Network (DQN)

```
class FullyConnectedModel(torch.nn.Module):

    def __init__(self, input_size, output_size):
        super().__init__()

        self.linear1 = torch.nn.Linear(input_size, 16)
        self.activation1 = torch.nn.ReLU()
        self.linear2 = torch.nn.Linear(16, 16)
        self.activation2 = torch.nn.ReLU()
        self.linear3 = torch.nn.Linear(16, 16)
        self.activation3 = torch.nn.ReLU()

        self.output_layer = torch.nn.Linear(16, output_size)
        #no activation output layer

        #initialization
        torch.nn.init.xavier_uniform_(self.linear1.weight)
        torch.nn.init.xavier_uniform_(self.linear2.weight)
        torch.nn.init.xavier_uniform_(self.linear3.weight)
        torch.nn.init.xavier_uniform_(self.output_layer.weight)

    def forward(self, inputs):
        x = self.activation1(self.linear1(inputs))
        x = self.activation2(self.linear2(x))
        x = self.activation3(self.linear3(x))
        x = self.output_layer(x)
        return x
```

DQN architecture is already provided to you!

Part 2: Q-Network

```python
class QNetwork():

    # This class essentially defines the network architecture.
    # The network should take in state of the world as an input,
    # and output Q values of the actions available to the agent as the output.

    def __init__(self, env, lr, logdir=None):
        # Define your network architecture here. It is also a good idea to define any training operations
        # and optimizers here, initialize your variables, or alternately compile your model here.
        # TODO Implement this method

        pass

    def save_model_weights(self, suffix):
        # Helper function to save your model / weights.
        path = os.path.join(self.logdir, "model")
        torch.save(self.model.state_dict(), model_file)
        return path

    def load_model(self, model_file):
        # Helper function to load an existing model.
        return self.model.load_state_dict(torch.load(model_file))

    def load_model_weights(self,weight_file):
        # Optional Helper function to load model weights.
        pass
```

Init: environment, learning rate, QNetwork (what are the input and output size of the model?), Adam optimizer, logging directory (if you want to use it)

Everything else is optional!

Part 2: Replay Memory

```
class Replay_Memory():

    def __init__(self, memory_size=50000, burn_in=10000):
        # The memory essentially stores transitions recorder from the agent
        # taking actions in the environment.

        # Burn in episodes define the number of episodes that are written into the
        memory from the
        # randomly initialized agent. Memory size is the maximum size after which old
        elements in the memory are replaced.
        # A simple (if not the most efficient) was to implement the memory is as a
        list of transitions.

        # Hint: you might find this useful:
        #       collections.deque(maxlen=memory_size)
        # TODO Implement this method
        pass

    def sample_batch(self, batch_size=32):
        # This function returns a batch of randomly sampled transitions - i.e. state,
        action, reward, next state, terminal flag tuples.
        # You will feed this to your model to train.
        # TODO Implement this method
        pass

    def append(self, transition):
        # Appends transition to the memory.
        # TODO Implement this method
        pass
```

Init: Initialize a replay buffer to store (states, actions, rewards, next_states, termination), memory size, burn in value (initial trajectory size after initialization).
Hint: use collections.deque(maxlen=memory_size)

Sample_batch: samples a random batch from the memory (make sure it works, otherwise your entire algorithm will break)

Append: add a (state, action, reward, next_state, termination) to the replay memory
Warning: if you don't use a deque with a max length, then you will need to manually keep track of the length to avoid your code from running too slow

Part 2: DQN Agent

```
class DQN_Agent():

    # In this class, we will implement functions to do the following.
    # (1) Create an instance of the Q Network class.
    # (2) Create a function that constructs a policy from the Q values predicted by the Q Netwo
    #       (a) Epsilon Greedy Policy.
    #       (b) Greedy Policy.
    # (3) Create a function to train the Q Network, by interacting with the environment.
    # (4) Create a function to test the Q Network's performance on the environment.
    # (5) Create a function for Experience Replay.

    def __init__(self, environment_name, render=False):

        # Create an instance of the network itself, as well as the memory.
        # Here is also a good place to set environmental parameters,
        # as well as training parameters - number of episodes / iterations, etc.
        # TODO Implement this method
        pass

    def epsilon_greedy_policy(self, q_values):
        # Creating epsilon greedy probabilities to sample from.
        # TODO Implement this method
        pass

    def greedy_policy(self, q_values):
        # Creating greedy policy for test time.
        # TODO Implement this method
        pass

    def train(self):
        # In this function, we will train our network.

        # When use replay memory, you should interact with environment here, and store these
        # transitions to memory, while also updating your model.
        # TODO Implement this method
        pass

    def test(self, model_file=None):
        # Evaluate the performance of your agent over 20 episodes, by calculating average
        cumulative rewards (returns) for the 20 episodes.
        # Here you need to interact with the environment, irrespective of whether you are using
        replay memory.
        # TODO Implement this method
        pass

    def burn_in_memory(self):
        # Initialize your replay memory with a burn_in number of episodes / transitions.
        # TODO Implement this method
        pass
```

Init: initialize hyperparameters, replay memory, QNetwork, target network

Epsilon_greedy_policy: return an action based on the epsilon greedy policy (eps=0.05)

Greedy_policy: return an action based on the greedy policy

Train: take a step in the environment, add the observations to the replay memory, make a gradient update by sampling a batch from the replay memory

Test: interact with the environment to get your respective rewards

Burn_in_memory: initialize replay memory with burn_in number of transitions with random agent

Main: initialize the DQN agent and train for num_episodes. Make sure to keep track of the reward, plot the average rewards and the range
Hint: use plt.fill_between to plot the range (max, min rewards)