Solving OpenAI LunarLander-v2 Problem

Using Reinforcement Learning Algorithms

Aman Arora

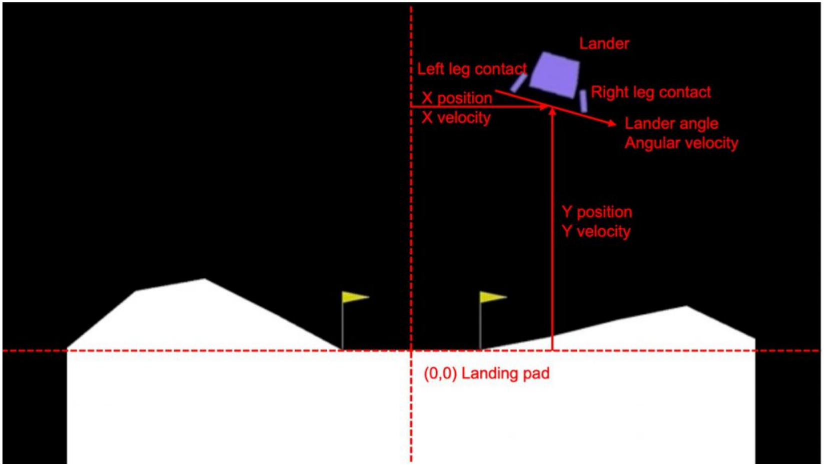
mt5110585@iitd.ac.in

Department of Mathematics

Indian Institute of Technology, Delhi

July 6, 2021

Table of Contents


Lander
Left leg contact
Right leg contact
X position
X velocity
Lander angle
Angular velocity
Y position
Y velocity
(0,0) Landing pad

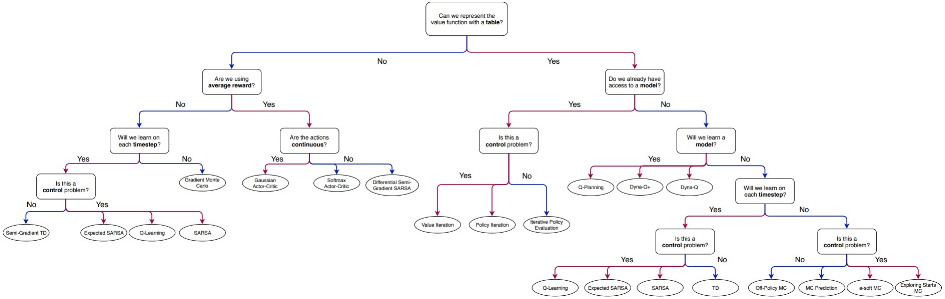
States, Actions and Rewards

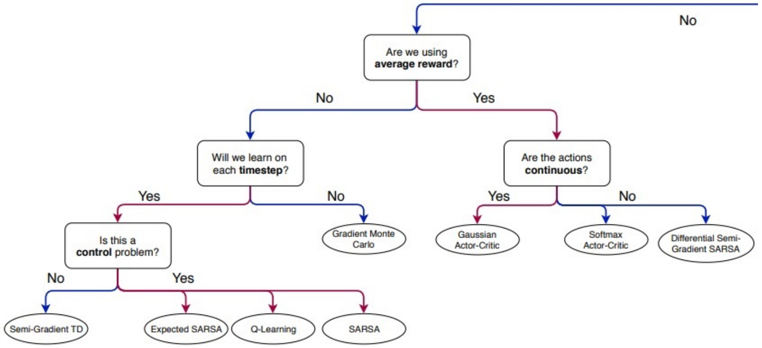
- **States** ✓
- **Actions**:
  - All engines disabled
  - Enable left engine
  - Enable main engine
  - Enable right engine

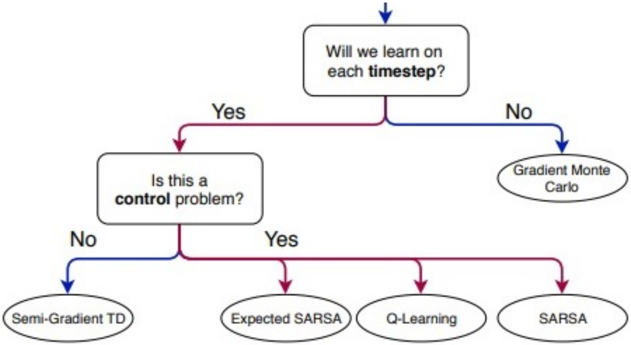
- **Reward function**:
  - -0.3 for using main engine
  - -0.03 for using side engine
  - -100 for crashing
  - $+10$ for each leg contact
  - $[+100, +140]$ for successful landing

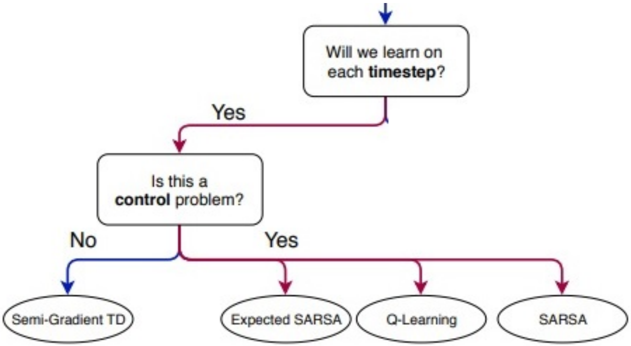
  Episode finishes if the lander crashes or comes to rest.

Choosing the learning algorithm


Can we represent the value function with a **table**?
No
Yes
Are we using **average reward**?
Do we already have access to a **model**?
No
Yes
Yes
No
Will we learn on each **timestep**?
Are the actions **continuous**?
Is this a **control** problem?
Will we learn a **model**?
Yes
No
Yes
No
Yes
No
Yes
No
Is this a **control** problem?
Gradient Monte Carlo
Gaussian Actor-Critic
Softmax Actor-Critic
Differential Semi-Gradient SARSA
Value Iteration
Policy Iteration
Iterative Policy Evaluation
Q-Planning
Dyna-Q+
Dyna-Q
Will we learn on each **timestep**?
No
Yes
Semi-Gradient TD
Expected SARSA
Q-Learning
SARSA
Yes
No
Is this a **control** problem?
Is this a **control** problem?
Yes
No
No
Yes
Q-Learning
Expected SARSA
SARSA
TD
Off-Policy MC
MC Prediction
e-soft MC
Exploring Starts MC


No
Are we using **average reward**?
No
Yes
Will we learn on each **timestep**?
Are the actions **continuous**?
Yes
No
Yes
No
Is this a **control** problem?
Gradient Monte Carlo
Gaussian Actor-Critic
Softmax Actor-Critic
Differential Semi-Gradient SARSA
No
Yes
Semi-Gradient TD
Expected SARSA
Q-Learning
SARSA


Will we learn on each **timestep**?
Yes
No
Gradient Monte Carlo
Is this a **control** problem?
No
Yes
Semi-Gradient TD
Expected SARSA
Q-Learning
SARSA


Will we learn on each **timestep**?
Yes
Is this a **control** problem?
No
Yes
Semi-Gradient TD
Expected SARSA
Q-Learning
SARSA

Prediction vs Control

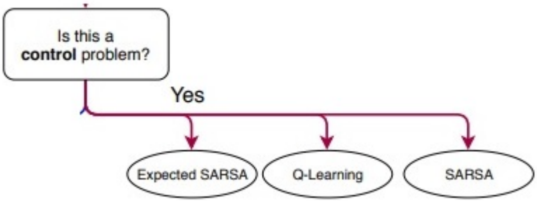
Recall:

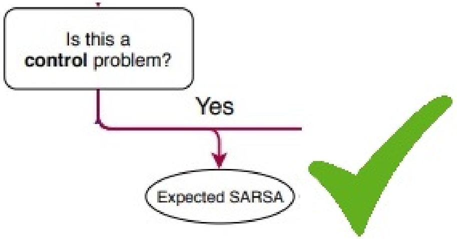
$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a')$$

and

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

- **Prediction**: the policy is supplied, and the goal is to measure how well it performs. That is, to predict the expected total reward from any given state assuming the function $\pi(a|s)$ is fixed.
- **Control**: the policy is not fixed, and the goal is to find the optimal policy. That is, to find the policy $\pi(a|s)$ that maximises the expected total reward from any given state.


Is this a **control** problem?
Yes
Expected SARSA
Q-Learning
SARSA


Is this a
control problem?
Yes
Expected SARSA

Temporal Difference Learning - for prediction

- **Goal**: learn v_π from experience under policy π
- Incremental every visit Monte-Carlo:
  - Update value $V(S_t)$ towards *actual* return G_t

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

- Simplest temporal-difference learning algorithm: TD(0)
  - Update value $V(S_t)$ towards *estimated* return $R_{t+1} + \gamma V(S_{t+1})$

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

  - $R_{t+1} + \gamma V(S_{t+1})$ is called the *TD target*
  - $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD error*
  - α is the step size

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t - V(S_t)\right)$$

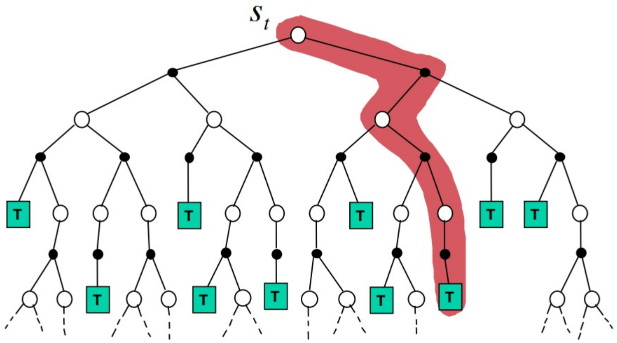


Figure: Backup diagram for Monte-Carlo

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

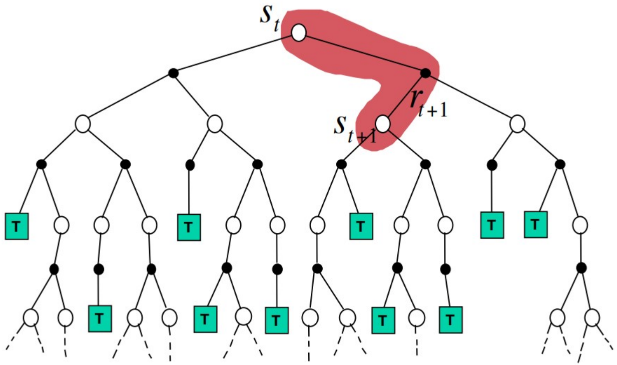


Figure: Backup diagram for TD

Advantages of TD over MC

1. TD can learn online after every step
2. TD can learn from incomplete sequences
3. TD works in continuing (non-terminating) environments

1. MC must wait until end of episode before return is known
2. MC can only learn from complete sequences
3. MC only works for episodic (terminating) environments

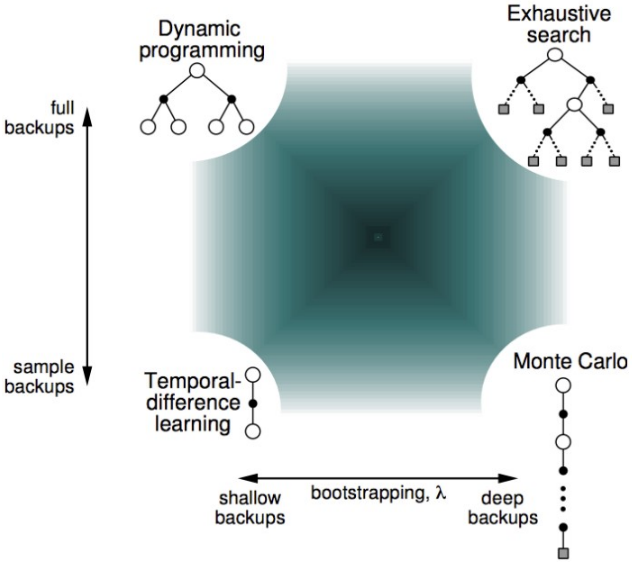
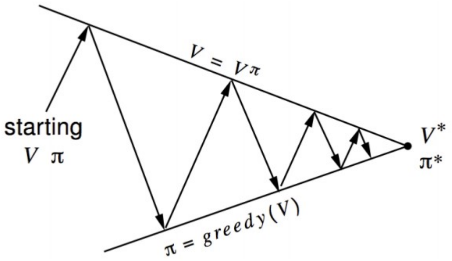


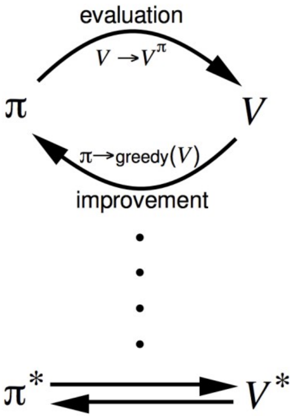
Figure: Unified view of Reinforcement Learning

Generalized Policy Iteration



Policy evaluation Estimate v_π
e.g. Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
e.g. Greedy policy improvement



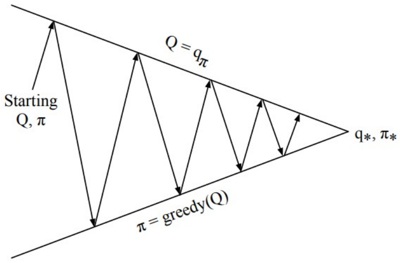
GPI with Action-Value Function (Model-Free)

- Greedy policy improvement over $V(s)$ requires model of MDP:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\arg\max}\, \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

- Greedy policy improvement over $Q(s,a)$ is model-free:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\arg\max}\, Q(s,a)$$



ϵ-greedy Exploration

- Simplest idea for ensuring continual exploration.
- All *m* actions are tried with non-zero probability.
- With probability $1 - \epsilon$ choose the greedy action.
- With probability ϵ choose an action at random

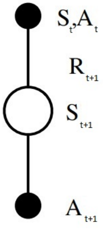
$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, Q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

TD for Control

Idea:

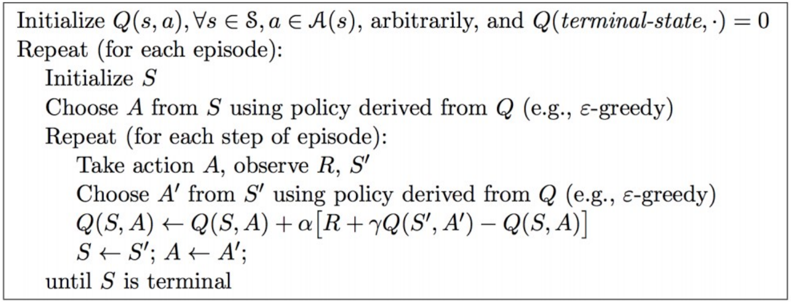
- Apply TD to $Q(S_t, A_t)$
- Use ϵ-greedy policy improvement
- Update at every time-step.

Performing these steps gives us our algorithm called - **SARSA**.



$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(r_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

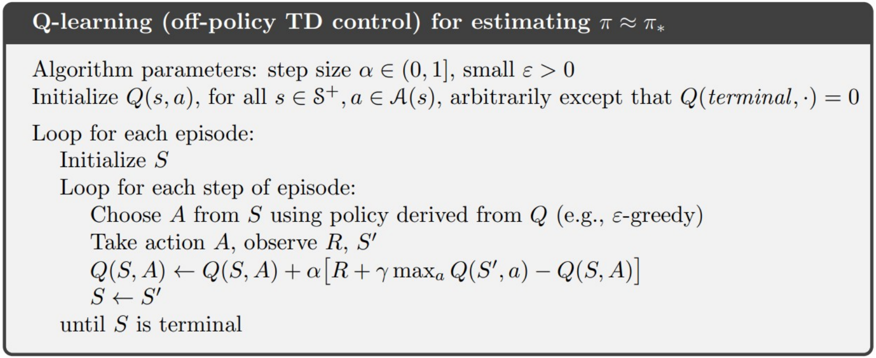
SARSA Algorithm



```
Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

Q-Learning: Off-policy TD Control

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



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Expected SARSA

- Sarsa's convergence guarantee requires that every state be visited infinitely often
- There can be substantial variance in Sarsa updates, since A_{t+1} is not selected deterministically and the behavior is typically stochastic to ensure sufficient exploration.
- Expected Sarsa exploits this knowledge to prevent stochasticity in the policy from further increasing variance by basing the update, not on $Q(S_{t+1}, A_{t+1})$, but on its expected value $\mathbb{E}_\pi[Q(S_{t+1}, A_{t+1})]$.

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}_\pi[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned}$$

Design Choices

We need to make some meta-parameter choices to implement our agent:

- Function Approximator
- Optimizer for updating the approximated action values
- How to incorporate exploration?

Value Function Approximation

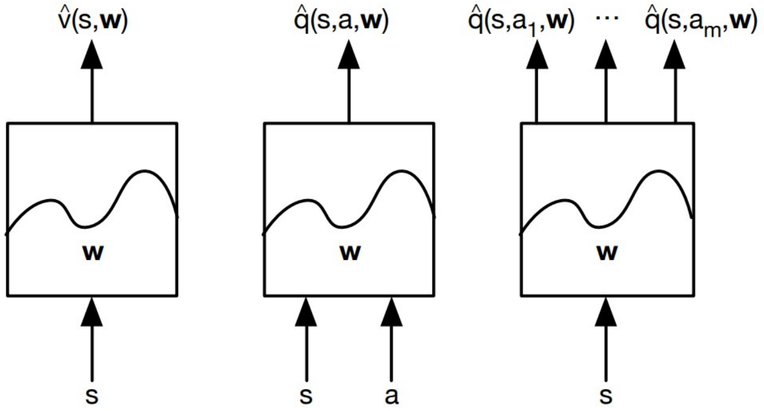
- So far we have represented value function by a lookup table.
  - Every state s has an entry $V(s)$.
  - Or every state-action pair (s, a) has an entry $Q(s, a)$.
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
- Solution for large MDPs:
  - Estimate value function with function approximation:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

$$\text{or} \qquad \hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a))$$

  - Generalise from seen states to unseen states.
  - Update parameter **w** using MC or TD learning.

Types of Value Function Approximation


$\hat{v}(s,\mathbf{w})$
$\hat{q}(s,a,\mathbf{w})$
$\hat{q}(s,a_1,\mathbf{w})$ ⋯ $\hat{q}(s,a_m,\mathbf{w})$
w
w
w
s
s
a
s

First option: Tile Coding

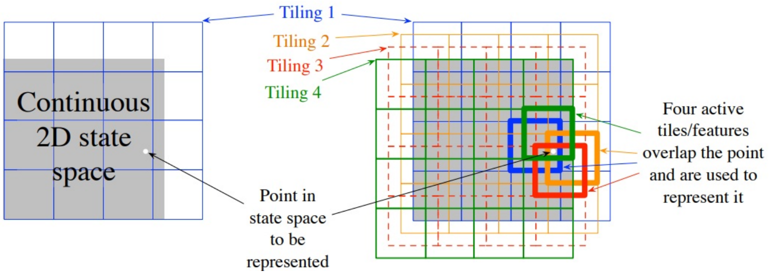
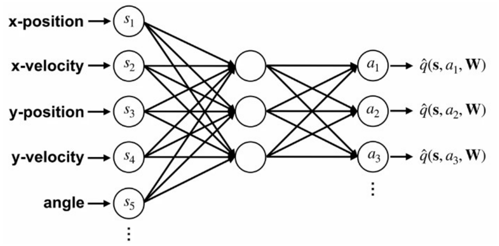
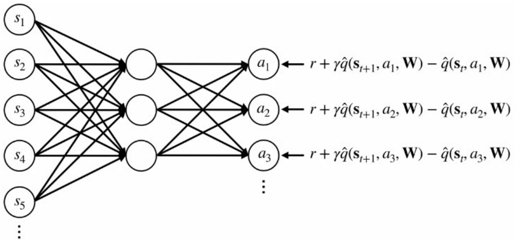

Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Figure: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

Better option: Neural Networks


x-position
x-velocity
y-position
y-velocity
angle
s_1
s_2
s_3
s_4
s_5
a_1
a_2
a_3
$\hat{q}(\mathbf{s}, a_1, \mathbf{W})$
$\hat{q}(\mathbf{s}, a_2, \mathbf{W})$
$\hat{q}(\mathbf{s}, a_3, \mathbf{W})$


s_1
s_2
s_3
s_4
s_5
a_1
a_2
a_3
$r + \gamma\hat{q}(\mathbf{s}_{t+1}, a_1, \mathbf{W}) - \hat{q}(\mathbf{s}_t, a_1, \mathbf{W})$
$r + \gamma\hat{q}(\mathbf{s}_{t+1}, a_2, \mathbf{W}) - \hat{q}(\mathbf{s}_t, a_2, \mathbf{W})$
$r + \gamma\hat{q}(\mathbf{s}_{t+1}, a_3, \mathbf{W}) - \hat{q}(\mathbf{s}_t, a_3, \mathbf{W})$

Choice of activation function

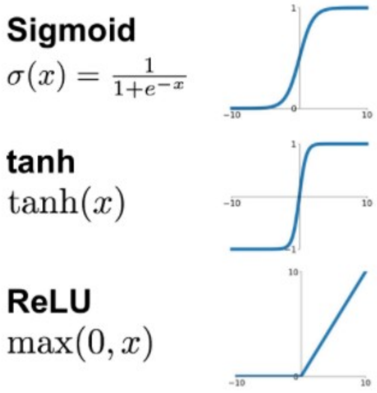
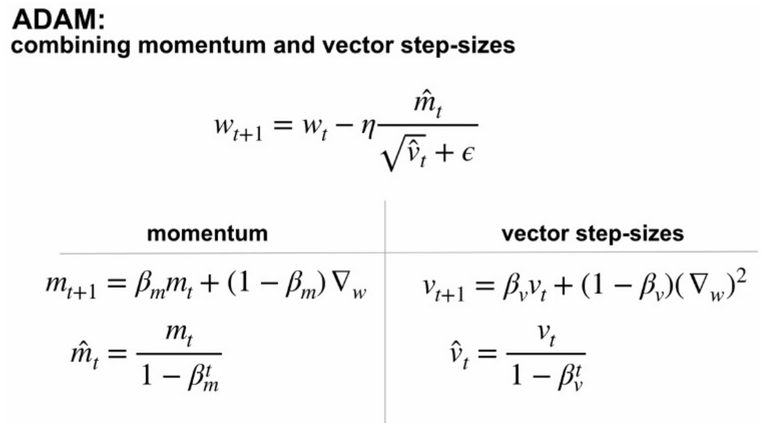


Figure: Some common activation functions

Optimizing the Neural Network - ADAM Algorithm



ADAM:
combining momentum and vector step-sizes

$$w_{t+1} = w_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

momentum	vector step-sizes
$m_{t+1} = \beta_m m_t + (1 - \beta_m) \nabla_w$	$v_{t+1} = \beta_v v_t + (1 - \beta_v)(\nabla_w)^2$
$\hat{m}_t = \frac{m_t}{1 - \beta_m^t}$	$\hat{v}_t = \frac{v_t}{1 - \beta_v^t}$

Exploration Method - Softmax function

- ϵ-greedy policy does not consider the individual action values when choosing an exploratory action in a state and instead chooses randomly when doing so.
- **Softmax policy** explores according to the action-values, meaning that an action with a moderate value has a higher chance of getting selected compared to an action with a lower value.

$$P(A_t = a | S_t = s) \doteq \frac{e^{Q(s,a)/\tau}}{\sum_{b\in A} e^{Q(s,b)/\tau}}$$

- Softmax policy exponentiates action values, if those values are large, exponentiating them could get very large. We therefore subtract the maximum action-value from the action-values.

$$P(A_t = a | S_t = s) \doteq \frac{e^{Q(s,a)/\tau - max_c Q(s,c)/\tau}}{\sum_{b\in A} e^{Q(s,b)/\tau - max_c Q(s,c)/\tau}}$$

Computing the TD-error

The basic idea is that we are looking to compute a form of a TD error. In order to so, we can take the following steps:

- compute the action-values for the next states using the action-value network Q_t,
- compute the policy $\pi(b|s')$ induced by the action-values Q_t (using the softmax function),
- compute the Expected-SARSA targets $r + \gamma \left(\sum_b \pi(b|s') Q_t(s', b)\right)$,
- compute the action-values for the current states using the latest Q_{t+1}, and,
- compute the TD-errors with the Expected-SARSA targets.

Optimizing the neural network

After we implemented `get_td_error()` function, we use it to implement the `optimize_network()` function. In this function, we:

- get the TD-errors vector from `get_td_error()`,
- make the TD-errors into a matrix using zeroes for actions not taken in the transitions,
- pass the TD-errors matrix to the `get_TD_update()` function of network to calculate the gradients times TD errors, and,
- perform an ADAM optimizer step.

The `agent()` function

After implementing the `optimize_network()` function, we can implement the agent, i.e., the `agent_step()` and `agent_end()` functions. We:

- select an action (only in `agent_step()`),
- add transitions (consisting of the state, action, reward, terminal, and next state) to the replay buffer, and,
- update the weights of the neural network by doing multiple replay steps and calling the `optimize_network()` function that we already implemented.

Results

- After implementing the agent, we use it to run an experiment on the LunarLander problem.
- We plot the learning curve of the agent to visualize learning progress.
- To plot the learning curve, we use the sum of rewards in an episode as the performance measure.

the `run_experiment()` method

```
    # Experiment parameters
experiment_parameters = {
    "num_runs" : 1,
    "num_episodes" : 3000,
    # OpenAI Gym environments allow for a timestep limit
    timeout, causing
    # episodes to end after some number of timesteps.
    # Here we use the default of 1000.
    "timeout" : 1000
}

# Environment parameters
environment_parameters = {}

current_env = LunarLanderEnvironment

# Agent parameters
agent_parameters = {
    'network_config': {
        'state_dim': 8,
        'num_hidden_units': 256,
        'num_actions': 4
    },
```

the run_experiment() method

```
    'optimizer_config': {
        'step_size': 1e-3,
        'beta_m': 0.9,
        'beta_v': 0.999,
        'epsilon': 1e-8
    },
    'replay_buffer_size': 50000,
    'minibatch_sz': 8,
    'num_replay_updates_per_step': 4,
    'gamma': 0.99,
    'tau': 0.001
}
```

code/run_experiment().py

Results

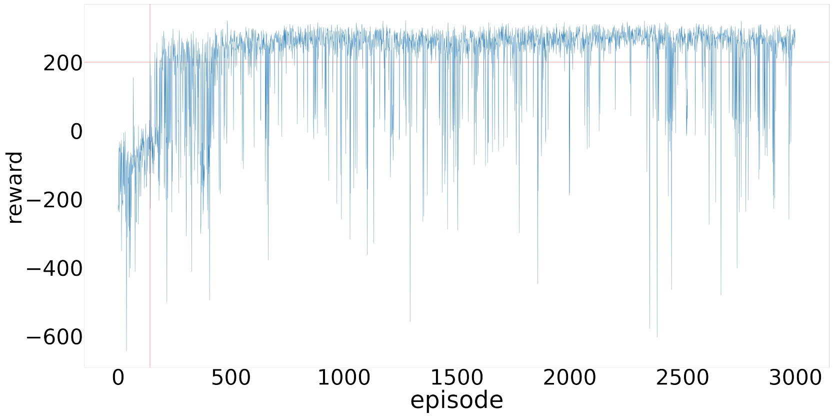


Figure: Learning curve for 3000 episodes

Results - after mean smoothing

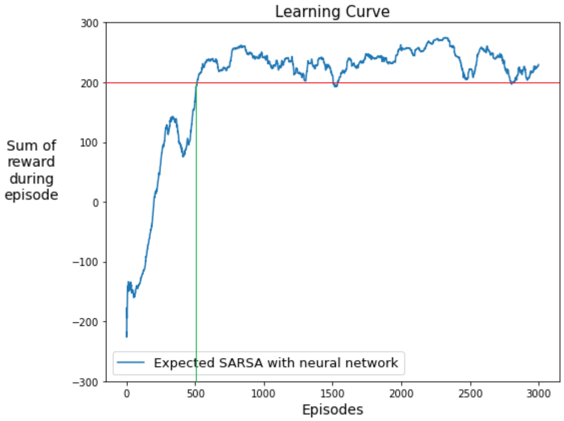


Figure: Learning curve for 3000 episodes

Parameter Study - step size η

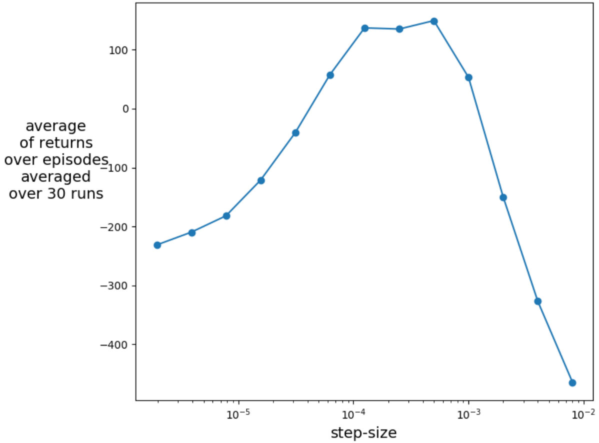


Figure: Parameter study for various step-sizes

Further unexplored work

- Generating `.mp4` visual video file for this agent
- Although we achieve a reward above 200 in episode 141 for the first time, and quite consistently after episode 500, still there are multiple episodes where performance of our agent drops. We need to fix this performance before we can submit to the leaderboard.
- Hyperparameter tuning for various other hyperparameters such as β_m, β_v, offset ϵ, number of hidden layers, τ, γ. This is restricted by the availability of computation power (I wish I had an NVIDIA TITAN V.)
- We can also try to solve another version of this problem with some added uncertainties like **noisy observations**, **random engine failure**, **random forces from solar winds**, etc. as mentioned in a paper by *S.Gadgil, Y.Xin, C. Xu*

Thank You !