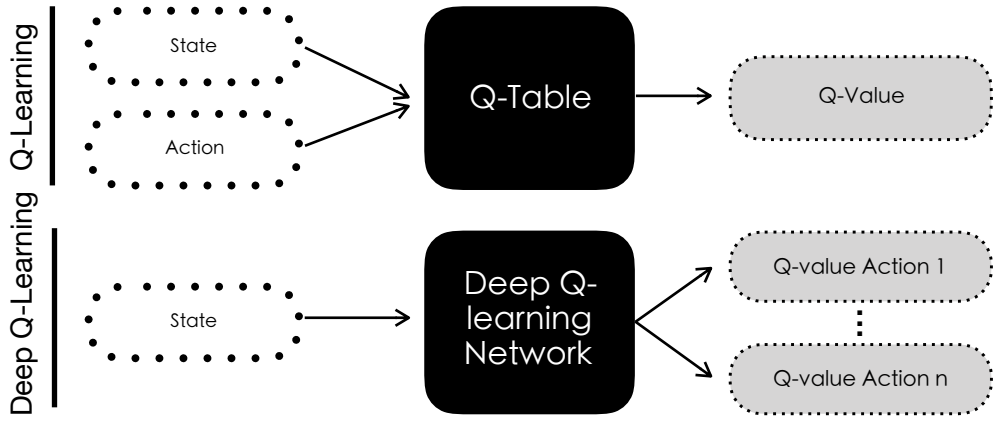
1. Write and describe the deep Q-network (DQN) algorithm. How does DQN address the issue of the deadly triad when Q-learning integrates with deep learning, i.e., the combination of off-policy learning, function approximation, and bootstrapping? (4%)

Leveraging Deep Q-learning (DQN) become necessary when one realises that producing and updating a Q-table can become challenging and ineffective in large state space environments, in other words, Q-learning is not scaleable. DQN is a result of implementing a Neural Network which will be responsible for taking states and assign Q-values for each action in the particular state.



Before the development of DQN cracking the deadly triad, off-policy learning: problem is a result of the max operator in the Bellman update in Q-learning, scalable function approximation and bootstrapping meaning the algorithm creates a estimation based on previous approximation by the model. DNQ stabilises the learning process by:

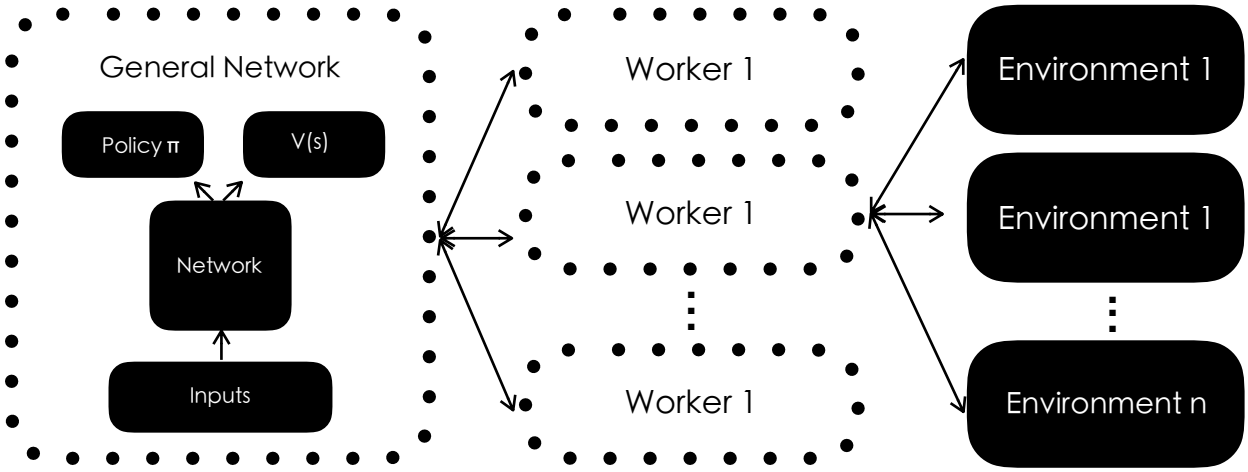
1. Remapping the Temporal Difference error to a fixed interval of [-1,1]
2. Experience replay, meaning that the tuple -state, action, next-state and reward- are not directly used to update the Q-function. Instead, it is updated by sampling a random batch of memory tuples at each time-step.
3. Replacing one single network by two separate networks, one online network parametrised by w^- to generate Q-values for the epsilon greedy action selection and one target network to generate targets. Weights of the target network is updated by the weights of online network at each time-step.

More to this point, DQN can also be interpreted a form of Q-Learning with embedded function approximation. This means that DQN algorithm tries to learn a state-action value Q-function through minimising temporal-difference errors. For instance DQN tries to make the value of Q(s,a) and $r+\gamma$ maxa' Q(s',a') closer, this procedure is mainly done using a greedy or epsilon-greedy policy. In other words, the Q-function generated by DQN, takes the input state and action and generates the reward expected for the pair , the best action for that particular state will be where Q-function is maximum.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

4. Write and describe the A3C algorithm. Why A3C does not need experience replay to stabilize the training? For Problem 1 and 2, you can copy the pseudocode. (3%)

The Asynchronous Advantage Actor-Critic (A3C) algorithm in Tensorflow, due to the speed, simplicity and robustness of execution, has in some ways replaced the DQN algorithm. DQN is based on single neural network which interacts with single environment, conversely, A3C consists of multiple networks working parallel governed by a general network which consequently, omits the necessity of experience reply.



The A3C algorithm has three features:

Asynchronous: In contrast to DQN, where one network is implemented in single environment, A3C leverages multiple networks and environments in an interactive manner. Each environment represents an agent with their unique copy of the global network. Existence of several networks in A3C, creates a stable loop in which experience is shared and network, therefore, omitting Experience Reply which is essential to stabilise single-network algorithms such as DQN.

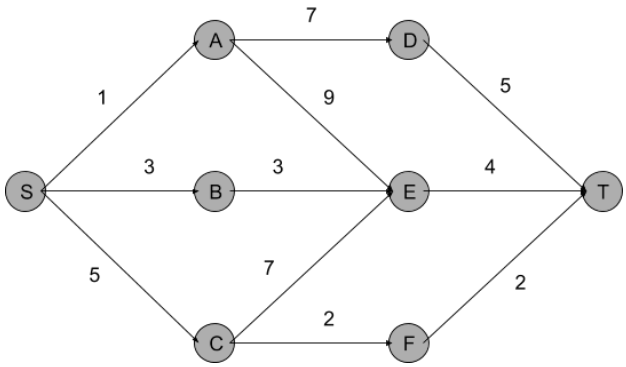
Advantage: This attribute of the A3C network applies to using advantage approximation in the network which allows the agent to not only determine value of an action but also comparison of the performance of the action to what the model expected it to be. This function allows the model to determine the low-performance parts of the model.

Actor-Critic: This feature combines the two methods, value-iteration(Q-Learning) and Policy Gradient. Two factors are considered in A3C algorithm, value function(V) and Policy (π). Value functions(value of a state) and policies(possibility of actions) are both calculated separately in the network while the critic agent uses the values to update the policies.

global shared parameter vectors $\boldsymbol{\theta}$ and $\boldsymbol{\theta}_v$, thread-specific parameter vectors $\boldsymbol{\theta}'$ and $\boldsymbol{\theta}'_v$
global shared counter $T = 0, T_{max}$
initialize step counter $t \leftarrow 1$
for $T \leq T_{max}$ do
  reset gradients, $d\boldsymbol{\theta} \leftarrow 0$ and $d\boldsymbol{\theta}_v \leftarrow 0$
  synchronize thread-specific parameters $\boldsymbol{\theta}' = \boldsymbol{\theta}$ and $\boldsymbol{\theta}'_v = \boldsymbol{\theta}_v$
  set $t_{start} = t$, get state s_t
  for s_t *not terminal and* $t - t_{start} \leq t_{max}$ do
    take a_t according to policy $\pi(a_t|s_t;\boldsymbol{\theta}')$
    receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1, T \leftarrow T + 1$
  end
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \boldsymbol{\theta}'_v) & \text{otherwise} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ do
    $R \leftarrow r_i + \gamma R$
    accumulate gradients wrt $\boldsymbol{\theta}'$: $d\boldsymbol{\theta} \leftarrow d\boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}'} \log \pi(a_i|s_i;\boldsymbol{\theta}')(R - V(s_i;\boldsymbol{\theta}'_v))$
    accumulate gradients wrt $\boldsymbol{\theta}'_v$: $d\boldsymbol{\theta}_v \leftarrow d\boldsymbol{\theta}_v + \nabla_{\boldsymbol{\theta}'_v}(R - V(s_i;\boldsymbol{\theta}'_v))^2$
  end
  update asynchronously $\boldsymbol{\theta}$ using $d\boldsymbol{\theta}$, and $\boldsymbol{\theta}_v$ using $d\boldsymbol{\theta}_v$
end

5. Write a program in Python to find the shortest path from node S to node T in the following graph, using model-free approaches: Monte Carlo (MC), SARSA, and Q-learning. The graph illustrates the nodes, edges, and weights on edges. Describe your approach as documentation/comments to your code, and draw figures for learning progress with largest action value function for the source state/node vs. episodes. (7%)

In the beginning, set action value function Q(.,.) as 0. Both the state space and the action space are {S, A, B, C, D, E, F, T}, the same in this particular problem. Set γ as 1. Try various α, e.g., 0.01, 0.05, 0.1. Try various ε, e.g., 0.01, 0.05, 0.1. Reward is negative of link weight. You can use a large number of iterations, e.g., 10000, for convergence of the algorithm. An episode terminates when the destination is reached.



Here is a simple introduction to how the codes were developed.

Q-Learning Algorithm:

Q-Learning algorithm is an off-policy based on maximum value in Q-Table.

Q-learning: An off-policy TD control 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
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-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

SARSA(state-action-reward-state-action) Algorithm:

Conversely, SARSA is an on-policy algorithm for TD learning. What makes Q-learning and SARSA different is that SARSA each state is determined by a new action and reward similar to the first state action and reward. However, in Q-Learning as mentioned the maximum reward is necessary for the new state action determination. Here, (s,a) stand for the original state and (s',a') represent the new state and action.

Sarsa: An on-policy TD control 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., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-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

Monte Carlo Algorithm:

This algorithm is a source-restricted algorithm which returns the output linked with the highest probability. In this algorithm the error margin plays and important part in determining the solution estimated by the model, therefore a solution proposed by the model could vary according to the assigned error margin.

First-visit MC policy evaluation (returns $V \approx v_\pi$)

Initialize:
  $\pi \leftarrow$ policy to be evaluated
  $V \leftarrow$ an arbitrary state-value function
  $\textit{Returns}(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:
  Generate an episode using π
  For each state s appearing in the episode:
    $G \leftarrow$ return following the first occurrence of s
    Append G to $\textit{Returns}(s)$
    $V(s) \leftarrow \text{average}(\textit{Returns}(s))$