University of Illinois Fall 2019

ECE 544NA: Pattern Recognition
Lecture 27: December 3

Lecturer: Alexander Schwing Scribe: Bingzhe Wei

1 Goals of This Lecture

- Extending the framework of Markov Decision Problems
- Getting to know Q-learning

2 A Review of Reinforcement Learning and Markov Decision Problems

2.1 Overview

In general, **Reinforcement learning** (RL) is concerned with enabling an **agent** acting in an **environment** to maximize the **cumulative reward** received by the agent from the aforementioned environment.

To formalize RL, we utilize the framework of Markov Decision Processes (MDPs). **Reinforcement learning for MDPs** is concerned with enabling an **agent** acting in an **stochastic Markovian environment** to maximize the **expected future reward** received by the agent from the aforementioned environment.

2.2 Problem Formulation - Informal

Let an agent be defined as an autonomous entity that receives observations and rewards from and acts upon an environment, and an environment be defined as that in which an agent exists in and which transitions between different states and provides an agent with observations and rewards which may depend on which actions are performed by the aforementioned agent.

Given the definitions above, suppose that there exists an environment E in which an agent exists, and assume that initially, i.e. when $t = 0$, E is in state s_0 and the agent has received observation o_0 and reward r_0. Subsequently, for each time step t, the agent performs action a_t, which causes the environment to transition to state s_t and results in the agent receiving observation o_t and reward r_t. This process is illustrated in Figure 1 below.

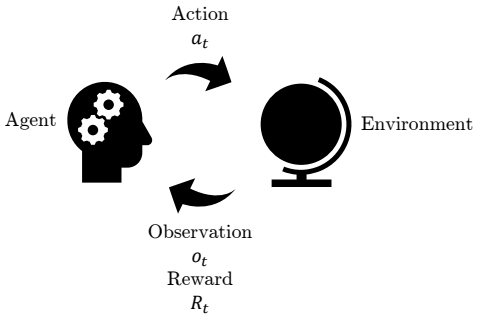

Figure 1: The Setting of Reinforcement Learning

We may then now define reinforcement learning as determining how should the agent act to maximize the cumulative reward received by the agent from E.

2.3 Characteristics of Environments

RL may be conducted under various settings, which differ depending on the properties of the environment. In particular, the environment may be either fully observable or partially-observable and may be either deterministic or stochastic.

Whereas the state of an environment at a particular time fully specifies that environment at that time, an observation of an environment at a particular time may hide certain information regarding that environment at that time. Without loss of generality, the observation received by an agent which exists in an environment after taking an action is a function of the state of that environment after being acted upon.

In a deterministic environment, state transitions are deterministic, i.e. the subsequent environment state is a deterministic function of the history of all past environment states and agent actions. In contrast, in a stochastic environment, state transitions are stochastic, i.e. the subsequent environment state is a distribution over some subset of all possible environment states given the history of all past environment states and agent actions. While stochasticity may be an inherent property of an environment, often stochasticity arises due to partial observability.

To illustrate these potential characteristics of the environment, let us consider `GridWorld`, illustrated below.

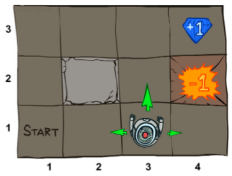

Figure 2: GridWorld

In `GridWorld`, the agent is a robot which moves about in an simulated environment consisting of a set of cells laid out in a 3×4 grid. The robot is initially in the lower left corner of the environment (Cell `(1,1)`). If the robot moves to a cell marked with a number, the robot receives a corresponding reward. The robot cannot move out of bounds or to Cell `(2,2)`, and the simulation terminates when the robot receives a reward. The objective here is then to decide how should the robot move in order to maximize its cumulative received reward.

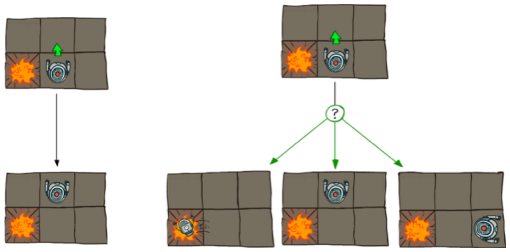

Figure 3: A Deterministic Environment (left) vs. A Stochastic Environment (right)

In `GridWorld`, the robot receives its current location after moving. Assuming that robot movement is deterministic and is influenced only by the robot's actions, knowing the layout of `GridWorld` and the current robot location is sufficient to fully specify `GridWorld`, and hence under such assumptions `GridWorld` is fully observable and deterministic. However, under more realistic assumptions, robot movement may be influenced by other factors which are not observed by the robot, such as wind. Under these assumptions, `GridWorld` is partially-observable and stochastic, since the robot may move left even if the robot desired to go up, as illustrated in Figure 3 above.

2.4 Problem Formulation - Formal

Although in general the environment and/or time may be continuous and/or discrete, in this lecture, we only consider scenarios where both the environment and time are discrete. More specifically, we will formalize RL for agents acting in discrete environments which are fully observable, discrete-time, stochastic, and Markov using the framework of Markov Decision Processes.

Formally, a Markov Decision Process (MDP) is defined by

- A set of states $s \in \mathcal{S}$
- A (state-dependent) set of actions $a \in \mathcal{A}_s$
- Transition probabilities $P(s' \mid s, a)$
- A reward function $R(s, a, s')$
- A start state $s_0 \in \mathcal{S}$
- Optionally a (set of) terminal/goal state(s) $s_G \in \mathcal{G} \subseteq \mathcal{S}$

As shown below, we may represent an MDP as a graph.

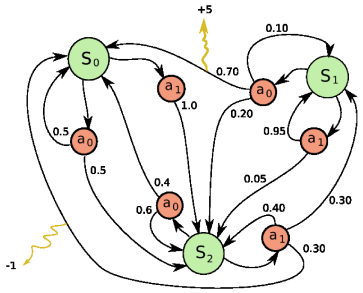

Figure 4: A Graphical Representation of a Markov Decision Process

In an MDP, due to the Markov assumption, the transition probabilities and rewards are decoupled from the past given the present. In particular, the following equation regarding transition probabilities holds.

$$\begin{aligned} P(S_{t+1} = s' \mid S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, A_{t-1} = a_{t-1}, \ \ldots \ , S_0 = s_0) \\ = P(S_{t+1} = s' \mid S_t = s_t, A_t = a_t) \end{aligned} \tag{1}$$

We now define a policy $\pi(s) : \mathcal{S} \mapsto \mathcal{A}_s$ to be a function mapping from states to actions that encodes how should an agent act when the environment is in a particular state. Hence, the value function of π, $V^\pi(s) : \mathcal{S} \mapsto \mathbb{R}$, is the expected future reward that will be received by an agent which follows π given that the environment is currently in state s.

We may then now define RL as determining the *optimal* policy $\pi^*(s)$ which encodes how should the agent act to maximize the expected future reward $V^{\pi^*}(s_0)$ received by the agent from the environment of the MDP.

2.5 RL vs. Discriminative Learning/Generative Learning

While the objectives of discriminative learning and generative learning are distinct, as discriminative learning seeks to learn a model that maximizes the conditional probability of labels given data from a dataset, i.e. $p(y|x)$, whereas generative learning aims to learn a model that maximizes the probability of data from a dataset, i.e. $p(x)$, the objectives of both discriminative learning and generative learning are based on the maximum likelihood principle.

In contrast, in RL, temporal correlations are particularly important. Decisions made earlier influence decisions made later as the state of the environment subsequent to an action being taken depends on which action was taken, and thus agents for RL must take historical context into account. There is no supervisor and only a reward signal, and whereas a supervisor is effectively an oracle, in RL, receiving a reward after performing an action may not necessarily indicate that the aforementioned action was optimal. While making decisions in a supervised setting results in immediate feedback, in RL, an agent may make decisions and not get immediate feedback.

Additionally, often a dataset does not exist, and an agent may be required to learn while interacting and observing an environment. However, the actions of the agent directly affect the data collected by the agent. These differences imply that techniques for learning in discriminative and generative settings are not directly applicable to learning in a RL setting.

3 Policy Evaluation

3.1 Overview

Policy evaluation is the process of determining the value function $V^\pi(s)$ of a policy π. To compute $V^\pi(s)$, we note that $V^\pi(s) = 0$ for any $s \in \mathcal{G}$, since an agent cannot act after reaching a terminal state, and that $V^\pi(s)$ for any $s \notin \mathcal{G}$ may be defined recursively as $V^\pi(s) = E_{s'}\left[R\left(s, \pi(s), s'\right) + V^\pi(s')\right]$, and $V^\pi(s)$ therefore satisfies the following equations.

$$V^\pi(s) = 0 \qquad s \in \mathcal{G} \tag{2}$$

$$V^\pi(s) = \sum_{s' \in \mathcal{S}} P\left(s' \mid s, \pi(s)\right)\left[R\left(s, \pi(s), s'\right) + V^\pi(s')\right] \qquad s \notin \mathcal{G} \tag{3}$$

Hence, in general, solving a linear system is required to compute $V^\pi(s)$. As solving a linear system is computationally expensive, we may alternatively compute $V^\pi(s)$ iteratively by using the following algorithm.

Algorithm 1: Iterative Policy Evaluation

$i = 0$
foreach $s \in \mathcal{S}$ **do**
  Initialize $V_0^\pi(s)$.
end
while not all $V_i^\pi(s)\ \forall s \in \mathcal{S}$ have converged **do**
  foreach $s \in \mathcal{S}$ **do**
    $V_{i+1}^\pi(s) = \sum_{s' \in \mathcal{S}} P\left(s' \mid s, \pi(s)\right)\left[R\left(s, \pi(s), s'\right) + V_i^\pi(s')\right]$
  end
  $i = i + 1$
end
return $V_i^\pi(s)\ \forall s \in \mathcal{S}$

3.2 A Simple MDP

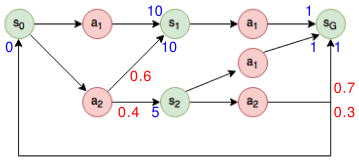

Figure 5: A Simple MDP. Unmarked transition probabilities are 1.0, and unmarked rewards are 0. (Transition Probabilities: Red, Rewards: Blue)

We now consider finding the value function for all of the possible policies (π^1, π^2, π^3) associated with the MDP illustrated in Figure 5 above. The policy graph associated with a policy is the restriction of an MDP to the states of the environment that an agent following the aforementioned policy will visit. For each policy, we first find the policy graph induced by that policy, then use both Equations 2 and 3 and Iterative Policy Evaluation to find the value function for that policy.

3.2.1 Policy π^1

Policy π^1 is defined as

$$\pi^1(s_0) = a_1, \pi^1(s_1) = a_1. \tag{4}$$

Figure 6: The Policy Graph Induced by π^1

Using Equations 2 and 3, we get the following linear system.

$$V^{\pi^1}(s_G) = 0 \tag{5}$$
$$V^{\pi^1}(s_1) = 1 \cdot \left[1 + V^{\pi^1}(s_G)\right] \tag{6}$$
$$V^{\pi^1}(s_0) = 1 \cdot \left[10 + V^{\pi^1}(s_1)\right] \tag{7}$$
$$\tag{8}$$

Hence, the solution is

$$V^{\pi^1}(s_G) = 0, V^{\pi^1}(s_1) = 1, V^{\pi^1}(s_0) = 11. \tag{9}$$

We note that $V^{\pi^1}(s_2)$ is undefined since $\pi^1(s_2)$ is not defined.
We now use iterative policy evaluation.

Iteration i	$V_i^{\pi^1}(s_0)$	$V_i^{\pi^1}(s_1)$	$V_i^{\pi^1}(s_G)$
0	0	0	0
1	$1 \cdot [10 + 0] = 10$	$1 \cdot [1 + 0] = 1$	0
2	$1 \cdot [10 + 1] = 11$	$1 \cdot [1 + 0] = 1$	0
3	$1 \cdot [10 + 1] = 11$	$1 \cdot [1 + 0] = 1$	0

As iterative policy evaluation has converged, the solution is

$$V^{\pi^1}(s_G) = 0, V^{\pi^1}(s_1) = 1, V^{\pi^1}(s_0) = 11. \tag{10}$$

Similarly, we note that $V^{\pi^1}(s_2)$ is undefined since $\pi^1(s_2)$ is not defined.

3.2.2 Policy π^2

Policy π^2 is defined as

$$\pi^2(s_0) = a_2, \pi^2(s_1) = a_1, \pi^2(s_2) = a_1. \tag{11}$$

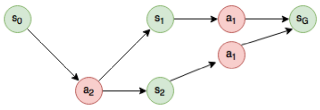

Figure 7: The Policy Graph Induced by π^2

Using Equations 2 and 3, we get the following linear system.

$$V^{\pi^2}(s_G) = 0 \tag{12}$$
$$V^{\pi^2}(s_2) = 1 \cdot \left[1 + V^{\pi^2}(s_G)\right] \tag{13}$$
$$V^{\pi^2}(s_1) = 1 \cdot \left[1 + V^{\pi^2}(s_G)\right] \tag{14}$$
$$V^{\pi^2}(s_0) = 0.6 \cdot \left[10 + V^{\pi^2}(s_1)\right] + 0.4 \cdot \left[5 + V^{\pi^2}(s_2)\right] \tag{15}$$
$$\tag{16}$$

Hence, the solution is

$$V^{\pi^2}(s_G) = 0, V^{\pi^2}(s_2) = 1, V^{\pi^2}(s_1) = 1, V^{\pi^2}(s_0) = 9. \tag{17}$$

We now use iterative policy evaluation.

Iteration i	$V_i^{\pi^2}(s_0)$	$V_i^{\pi^2}(s_1)$	$V_i^{\pi^2}(s_2)$	$V_i^{\pi^2}(s_G)$
0	0	0	0	0
1	$0.6 \cdot [10 + 0] + 0.4 \cdot [5 + 0] = 8$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	0
2	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 1] = 9$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	0
3	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 1] = 9$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	0

As iterative policy evaluation has converged, the solution is

$$V^{\pi^2}(s_G) = 0, V^{\pi^2}(s_2) = 1, V^{\pi^2}(s_1) = 1, V^{\pi^2}(s_0) = 9. \tag{18}$$

3.2.3 Policy π^3

Policy π^3 is defined as

$$\pi^3(s_0) = a_2, \pi^3(s_1) = a_1, \pi^3(s_2) = a_2. \tag{19}$$

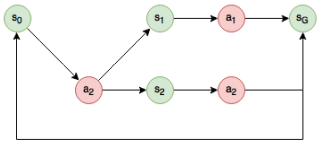

Figure 8: The Policy Graph Induced by π^3

Using Equations 2 and 3, we get the following linear system.

$$V^{\pi^3}(s_G) = 0 \tag{20}$$
$$V^{\pi^3}(s_2) = 0.7 \cdot \left[1 + V^{\pi^3}(s_G)\right] + 0.3 \cdot \left[0 + V^{\pi^3}(s_0)\right] \tag{21}$$
$$V^{\pi^3}(s_1) = 1 \cdot \left[1 + V^{\pi^3}(s_G)\right] \tag{22}$$
$$V^{\pi^3}(s_0) = 0.6 \cdot \left[10 + V^{\pi^3}(s_1)\right] + 0.4 \cdot \left[5 + V^{\pi^3}(s_2)\right] \tag{23}$$
$$\tag{24}$$

Hence, the solution is

$$V^{\pi^3}(s_G) = 0, V^{\pi^3}(s_2) = \frac{41}{11}, V^{\pi^3}(s_1) = 1, V^{\pi^3}(s_0) = \frac{111}{11}. \tag{25}$$

We now use iterative policy evaluation.

Iteration i	$V_i^{\pi^3}(s_0)$	$V_i^{\pi^3}(s_1)$	$V_i^{\pi^3}(s_2)$	$V_i^{\pi^3}(s_G)$
0	0	0	0	0
1	$0.6 \cdot [10 + 0] + 0.4 \cdot [5 + 0] = 8$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 0] = 0.7$	0
2	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 0.7] = 8.88$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 8] = 3.1$	0
3	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 3.1] = 9.84$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 8.88] = 3.36$	0

We see that as $i \to \infty$, the solution is

$$V^{\pi^3}(s_G) = 0, V^{\pi^3}(s_2) = \frac{41}{11}, V^{\pi^3}(s_1) = 1, V^{\pi^3}(s_0) = \frac{111}{11}. \tag{26}$$

4 Finding the Optimal Policy π^*

4.1 Overview

Given our definition of RL, policy π^a is superior to policy π^b if $V^{\pi^a}(s_0) > V^{\pi^b}(s_0)$. Hence, for any given policy π, $V^{\pi^*}(s_0) \geq V^{\pi}(s_0)$. In general, the optimal policy depends on the starting state s_0.

4.2 Exhaustive Search

The simplest method possible is exhaustive search. We first enumerate all possible policies, find the value function of each possible policy via policy evaluation, then pick a policy π^* which satisfies the condition that for any possible policy π, $V^{\pi^*}(s_0) \geq V^{\pi}(s_0)$.

Algorithm 2: Exhaustive Search

Enumerate all possible policies $\hat{\pi}$.
current_max_$V^{\pi}(s_0) = -\infty$
current_V^{π^*} = undefined
foreach $\pi \in \hat{\pi}$ **do**
    Perform policy evaluation on π to find $V^{\pi}(s_0)$.
    if current_max_$V^{\pi}(s_0) < V^{\pi}(s_0)$ **then**
        current_max_$V^{\pi}(s_0) = V^{\pi}(s_0)$
        current_$V^{\pi^*} = \pi$
    end
end
return current_V^{π^*}

Exhaustive search tends to be slow as policy evaluation is computationally expensive. Although it is possible to take advantage of common subgraphs present in the policy graphs induced by each policy to intelligently initialize $V_0^{\pi}(s)$ based on the results of previous iterative policy evaluation runs to enable iterative policy evaluation to run faster than policy evaluation based on solving linear systems, since there are at most $\prod_{s \in \mathcal{S}} |\mathcal{A}_s|$ policies, exhaustive search is not practical except when the environment has very few possible states.

For the MDP described in Section 3.2, there are three possible policies, which are π_1, π_2, and π_3 as described previously. Based on our computations in Section 3.2, since $V^{\pi^1}(s_0) > V^{\pi^2}(s_0)$ and $V^{\pi^1}(s_0) > V^{\pi^3}(s_0)$, the optimal policy for the MDP described in Section 3.2 is π_1.

4.3 Value Iteration

In value iteration, we search through the space of all value functions iteratively to find the value function of the optimal policy $V^*(s) = V^{\pi^*}(s)$, and then directly infer the optimal policy from $V^*(s)$. To do this, we utilize the Bellman Optimality Principle, which states that

$$V^*(s) = \max_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} P\left(s' \mid s, a\right) \left[R\left(s, a, s'\right) + V^*(s')\right]. \tag{27}$$

Given $V^*(s)$, the optimal policy is thus

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} P\left(s' \mid s, a\right) \left[R\left(s, a, s'\right) + V^*(s')\right]. \tag{28}$$

Whereas the value function of policy π, $V^\pi(s)$, gives the expected future reward when an agent follows policy π given that the environment is currently in state s, the Q-function of π, $Q^\pi(s, a)$, gives the expected future reward when an agent which follows policy π after taking action a given that the environment is currently in state s. Hence, the Q-function must satisfy

$$Q^\pi(s, a) = \sum_{s' \in \mathcal{S}} P\left(s' \mid s, a\right) \left[R\left(s, a, s'\right) + V^\pi(s')\right], \tag{29}$$

and we may alternatively define an analogous version of Equation 27 in terms of the Q-function of the optimal policy $Q^*(s, a) = Q^{\pi^*}(s, a)$ as

$$Q^*(s, a) = \sum_{s' \in \mathcal{S}} P\left(s' \mid s, a\right) \left[R\left(s, a, s'\right) + \max_{a' \in \mathcal{A}_{s'}} Q^*(s', a')\right]. \tag{30}$$

Given $Q^*(s, a)$, the optimal policy is thus

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}_s} Q^*(s, a). \tag{31}$$

While Equation 27 may be solved using linear program solvers, solving Equation 27 using linear program solvers is only practical when the environment has very few possible states. Instead, we consider solving Equation 27 iteratively. Value iteration typically scales better than exhaustive search but not to the sizes of environment state spaces often considered in contemporary MDPs.

Algorithm 3: Value Iteration

$i = 0$
foreach $s \in \mathcal{S}$ **do**
  Initialize $V_0^*(s)$.
end
while not all $V_i^*(s)$ have converged **do**
  foreach $s \in \mathcal{S}$ **do**
    $V_{i+1}^*(s) = \max_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} P\left(s' \mid s, a\right) \left[R\left(s, a, s'\right) + V_i^*(s')\right] = \max_{a \in \mathcal{A}_s} Q_i^*(s, a)$
  end
end
return
$\pi^*(s) = \operatorname{argmax}_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} P\left(s' \mid s, a\right) \left[R\left(s, a, s'\right) + V_i^*(s')\right] = \operatorname{argmax}_{a \in \mathcal{A}_s} Q_i^*(s, a) \; \forall s \in \mathcal{S}$

We now perform value iteration on the MDP described in Section 3.2.

Iteration i	$V_i^{\pi^*}(s_0)$	$V_i^{\pi^*}(s_1)$	$V_i^{\pi^*}(s_2)$	$V_i^{\pi^*}(s_G)$
0	0	0	0	0
1	10	1	1	0
2	11	1	3.7	0
3	11	1	4	0
4	11	1	4	0
5	11	1	4	0

Iteration i	$Q_i^{\pi^*}(s_0, a_1)$	$Q_i^{\pi^*}(s_0, a_2)$	$Q_i^{\pi^*}(s_1, a_1)$	$Q_i^{\pi^*}(s_2, a_1)$	$Q_i^{\pi^*}(s_2, a_2)$
0	0	0	0	0	0
1	$1 \cdot [10 + 0] = 10$	$0.6 \cdot [10 + 0] + 0.4 \cdot [5 + 0] = 8$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 0] = 0.7$
2	$1 \cdot [10 + 1] = 11$	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 1] = 9$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 10] = 3.7$
3	$1 \cdot [10 + 1] = 11$	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 3.7] = 10.08$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 11] = 4$
4	$1 \cdot [10 + 1] = 11$	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 4] = 10.2$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 11] = 4$
5	$1 \cdot [10 + 1] = 11$	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 4] = 10.2$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 11] = 4$

Since value iteration has converged, the optimal policy π^* is

$$\pi^*(s_0) = a_1, \pi^*(s_1) = a_1, \pi^*(s_2) = a_2. \tag{32}$$

4.4 Policy Iteration

In policy iteration, we search through the space of all policies by iteratively refining a policy π to maximize $V^\pi(s_0)$ by alternatively evaluating the policy and improving the policy.

Algorithm 4: Policy Iteration

Initialize π to a random policy.
while π has not converged **do**
  Perform policy evaluation on π to find $V^\pi(s)$.
  foreach $s \in \mathcal{S}$ **do**
    $\pi(s) = \text{argmax}_{a \in \mathcal{A}_s} \sum_{s' \in \mathcal{S}} P(s' \mid s, a) [R(s, a, s') + V^\pi(s')] = \text{argmax}_{a \in \mathcal{A}_s} Q^\pi(s, a)$
  end
end
return π

We now perform policy iteration on the MDP described in Section 3.2. Suppose π is initialized to π^2.

Iteration i	$V^\pi(s_0)$	$V^\pi(s_1)$	$V^\pi(s_2)$	$V^\pi(s_G)$
0	9	1	1	0
1	11	1	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 11] = 4$	0
2	11	1	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 11] = 4$	0

Iteration i	$Q^\pi(s_0, a_1)$	$Q^\pi(s_0, a_2)$	$Q^\pi(s_1, a_1)$	$Q^\pi(s_2, a_1)$	$Q^\pi(s_2, a_2)$
0	$1 \cdot [10 + 1] = 11$	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 1] = 9$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 9] = 3.4$
1	$1 \cdot [10 + 1] = 11$	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 4] = 10.2$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 11] = 4$
2	$1 \cdot [10 + 1] = 11$	$0.6 \cdot [10 + 1] + 0.4 \cdot [5 + 4] = 10.2$	$1 \cdot [1 + 0] = 1$	$1 \cdot [1 + 0] = 1$	$0.7 \cdot [1 + 0] + 0.3 \cdot [0 + 11] = 4$

Iteration i	$\pi(s_0)$	$\pi(s_1)$	$\pi(s_2)$
0	a_2	a_1	a_1
1	a_1	a_1	a_2
2	a_1	a_1	a_2

Since policy iteration has converged, the optimal policy π^* is

$$\pi^*(s_0) = a_1, \pi^*(s_1) = a_1, \pi^*(s_2) = a_2. \tag{33}$$

5 Q-Learning

5.1 Overview

In many applications of RL for MDPs, the transition probabilities and/or rewards of the MDP are unknown. Alternatively, we may wish to consider model-free RL. To deal with these scenarios, we consider directly approximating the Q-function from data by running a simulator to collect sample transitions (s, a, r, s') and then subsequently approximating the transition probabilities of the MDP using these samples, then subsequently extracting the optimal policy from the learnt Q-function.

5.2 Tabular Q-Learning

The Bellman Optimality Principle (Equation 30) suggests that a possible update rule to learn the Q-function from data is $Q(s, a) \approx r + \max_{a' \in \mathcal{A}_{s'}} Q(s', a')$ given a sample transition (s, a, r, s') from the environment. However, in our current setting, the transition probabilities in Equation 30 are unknown. To overcome this difficulty, we assume that the transition probabilities are reflected in how often we observe certain sample transitions. If a specific sample transition is observed very frequently, that sample transition should have a high transition probability, which is reflected by the Q-function values associated with that sample transition being updated more frequently. To enable the value of $Q(s, a)$ to smoothly transition to the value $r + \max_{a' \in \mathcal{A}_{s'}} Q(s', a')$ suggested by a sample transition (s, a, r, s'), we keep a running average of the previous values of $Q(s, a)$, which results in the following update rule.

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha \left(r + \max_{a' \in \mathcal{A}_{s'}} Q(s', a') \right), \tag{34}$$

where $\alpha \in (0, 1]$. The learnt policy may then be derived using the following equation.

$$\pi(s) = \operatorname*{argmax}_{a \in \mathcal{A}_s} Q(s, a) \tag{35}$$

To obtain sample transitions, an agent must act upon the environment. As the Q-function values associated with sample transitions that are not seen by an agent are not updated and are hence likely to be inaccurate, it is desirable for an agent to explore sample transitions associated with various combinations of environment states and actions. However, such exploration means that an agent may not discover state transitions associated with the optimal policy and hence cannot learn the optimal policy, as discovering state transitions associated with the optimal policy requires that an agent exploit the currently known most optimal policy derived using its current estimated Q-function values since exploration will likely require known sub-optimal actions to be selected and cause the agent to receive sub-optimal rewards.

We hence consider balancing between exploration and exploitation by selecting actions ϵ-greedily. In ϵ-greedy action selection, each time an action is selected by an agent, there is a $\epsilon \in [0, 1]$ chance of

selecting a random action, and $1-\epsilon$ chance of selecting an action based on the currently known most optimal policy derived using the agent's current estimated Q-function values. Typically, to ensure that estimated Q-function values converge, ϵ is initialized to $\epsilon_{initial} \in [0, 1]$ and is subsequently annealed following some schedule to a final value of $\epsilon_{final} \in [0, 1]$ which is less than $\epsilon_{initial}$.

As the algorithm described above estimates the value of each Q-function value independently, the algorithm described above may be implemented by storing the estimated Q-function values in a table, and hence the algorithm described above is known as Tabular Q-Learning.

Algorithm 5: Tabular Q-Learning

$\epsilon = \epsilon_{initial}$
foreach $s \in \mathcal{S}$ **do**
  foreach $a \in \mathcal{A}_s$ **do**
    Initialize $Q(s, a)$.
  end
end
while not all $Q(s, a)$ have converged **do**
  $\pi(s) = \text{argmax}_{a \in \mathcal{A}_s} Q(s, a)$
  Perform ϵ-greedy action and obtain a sample transition (s, a, r, s') from the environment.
  if $\epsilon \geq \epsilon_{final}$ **then**
    Reduce the value of ϵ according to some schedule.
  end
  $Q(s, a) = (1 - \alpha)Q(s, a) + \alpha \left(r + \max_{a' \in \mathcal{A}_{s'}} Q(s', a')\right)$
end
return $\pi(s) = \text{argmax}_{a \in \mathcal{A}_s} Q(s, a) \; \forall s \in \mathcal{S}$

The estimated Q-function values learnt using the Tabular Q-Learning algorithm after the robot has interacted with the environment of a stochastic version of `GridWorld` 1000 times is shown in Figure 9 below.

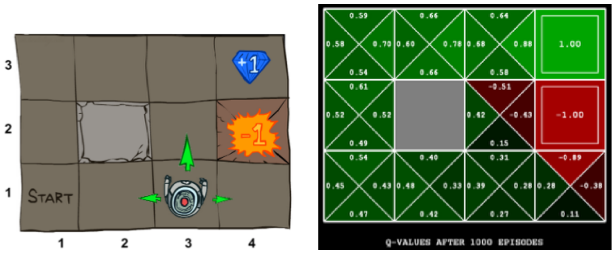

Figure 9: Tabular Q-Learning on a Stochastic Version of `GridWorld`

The policy learnt by the Tabular Q-Learning algorithm on `GridWorld` is shown in Figure 10 below. Note that the robot will move left if its current location is either Cell `(2,1)`, Cell `(3,1)`, Cell `(4,1)`, or Cell `(3,2)`, as such a policy minimizes the probability of the robot entering Cell `(4,2)`.

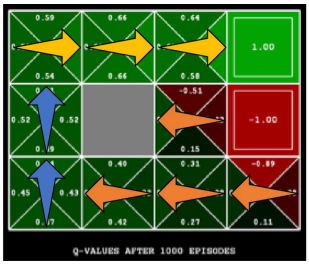

Figure 10: Policy Learnt by Tabular Q-Learning on a Stochastic Version of `GridWorld`

5.3 Deep Q-Learning

While Tabular Q-Learning works well when the environment has relatively few states and possible actions, Tabular Q-Learning estimates every Q-function value independently, and is hence not scalable to environments with many states and possible actions due to high storage requirements and slow convergence speed. Instead, we consider using function approximation to enable generalization across different Q-function values to enable faster convergence and to reduce storage requirements by storing only a relatively small set of parameters.

We now consider using RL to play Atari games.

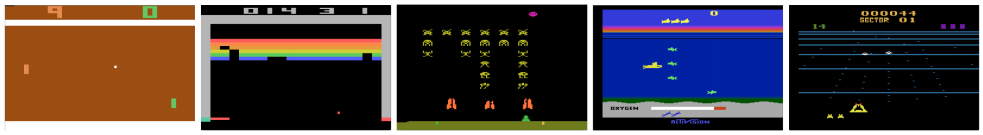

Figure 11: A Selection of Five Atari Games [7]

Atari games are typically played with an eight-directional digital joystick with a fire button. Hence, depending on the game, there are up to eighteen different actions available to an agent playing Atari games, as there are nine different possible joystick states and two different possible fire button states. The reward for an Atari-game playing agent is the score achieved by the agent through play.

An agent playing Atari games can only observe the screen. In Atari games, the current state of the screen does not fully specify the environment, as some aspect of the environment are controlled by the value of certain memory locations/registers that are not displayed on screen and are hence invisible to the agent. We can model such an environment using MDPs by considering Atari games to be fully observable but stochastic environments.

Typically, an Atari game session consists of multiple rounds of gameplay, with gameplay difficulty increasing after each round. A player starts with a fixed number of lives and an Atari game session terminates when all lives are lost regardless of the number of rounds that the player has completed. As the game state is reset after an Atari game session terminates, Atari games are episodic environments. Hence, a sample transition where s' corresponds to a state in which an Atari game session has terminated suggests that the target value is simply $y = r$, and the target value suggested by a sample transition is now

$$y = \left\{ \begin{array}{ll} r & \text{if } s' \text{ corresponds to a terminal state} \\ r + \max_{a'} Q(s', a') & \text{else} \end{array} \right\}, \tag{36}$$

Deep Q-Learning [7, 8] utilizes a deep neural network shown below in Figure 13 to approximate the Q-function and enable learning an agent to play multiple Atari games without game-specific hyperparameter tuning.

In practice, Q-Learning using function approximation is unstable due to correlations between sample transitions collected by the agent and correlations between the target value suggested by a sample transition (s, a, r, s') and the corresponding $Q(s, a)$. To ensure convergence, [8] use a buffer of recent sample transitions $\mathcal{D}$ and a separate target network to calculate the *discounted* target value

$$y = \left\{ \begin{array}{ll} r & \text{if } s' \text{ corresponds to a terminal state} \\ r + \gamma \max_{a'} Q(s', a') & \text{else} \end{array} \right\}, \tag{37}$$

where $\gamma \in (0, 1]$. The target network uses the same neural network architecture as the neural network that approximates Q-function values, but uses parameters from the neural network that approximates Q-function values during an earlier optimization iteration. [8] additionally use the Huber loss (loss gradient clipping) to further increase training stability.

The algorithm of [8] is shown below.

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 $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D
    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D
    Set $y_j = \left\{ \begin{array}{ll} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{array} \right.$
    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 12: The Algorithm of [8] (Deep Q-Learning)

While one possible neural network architecture would be a neural network that took as input both the current state and a possible action of the current state and outputs a single value, such a neural network architecture would require one forward pass per possible action of the current state. Alternatively, [7, 8] used a neural network that took as input only the current state and outputs a vector of values, one for every possible action in any state. With such a neural network architecture, only one forward pass is required regardless of how many actions are available to the agent in any state.

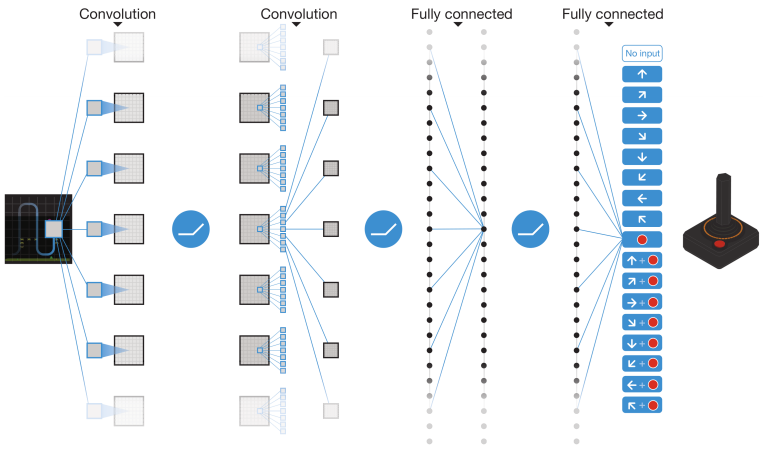

Figure 13: The Neural Network Function Approximator Used in [8]

In Figure 14 below, given the current game state (Game States 1, 2, and 3), based on its Q-function value estimates, an agent learnt using Deep Q-Learning (Green) playing Pong chooses to move its paddle upwards in order to strike the ball back towards the opposing player (Orange). Once the ball has moved past the opposing player (Game State 4), regardless of the action taken by the agent, the agent will gain a reward of one point. Hence, the Q-function value estimates of the game state of the rightmost frame for all possible actions are approximately equal.

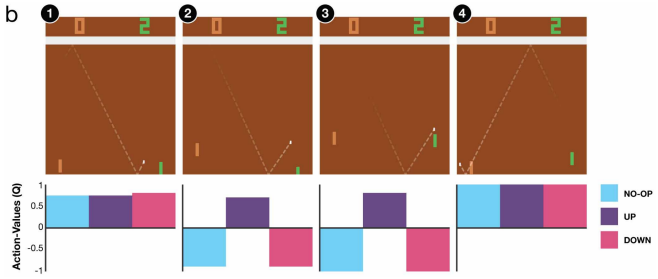

Figure 14: $Q(s, a)$ estimated by an Deep Q-Learning Agent Playing Pong [8]

The figure below shows the percentage difference between the score achieved by an agent trained using Deep Q-Learning and the score achieved by a professional human player on multiple Atari games. The trained agent is able to match human performance on many games and is able to achieve performance significantly higher than human performance on certain games.

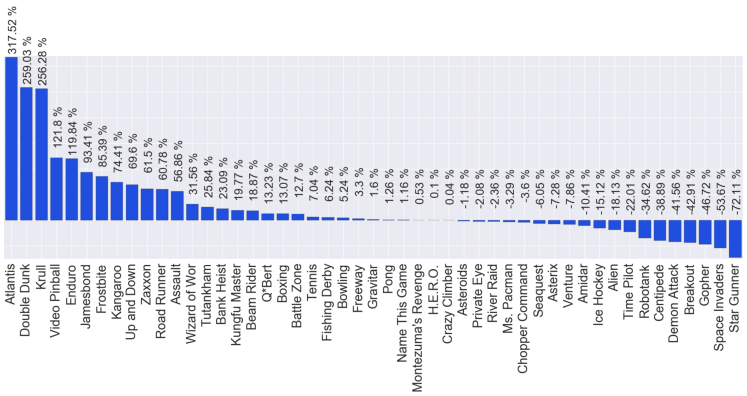

Figure 15: Deep Q-Learning on Atari Games, Achieved Score Percentage Difference vs. Professional Human Player

6 Summary

	To compute V^*, Q^*, or π^*:	To evaluate fixed policy π:
Known MDP	Value Iteration, Policy Iteration	Policy Evaluation
Unknown MDP (Model Free)	Q-Learning	Value Learning

Table 1: A Summary of RL Techniques

7 Project

7.1 Overview

In this project, we demonstrate how to implement Deep Q-Learning as described in [8]. Experimental results demonstrate that our implementation results in agents with comparable performance to that of [8] on Breakout and Pong. We also demonstrate that our implementation enables an agent to also play Enduro, River Raid, and Space Invaders without game-specific hyperparameter tuning.

7.2 Method

7.2.1 Algorithm

The algorithm of [8] was discussed previously in Section 5.3. We now discuss some practical considerations and resulting modifications to the algorithm discussed previously in Section 5.3. Frame preprocessing and environment modification transformations are performed in the order shown below in Figure 16 during training and in Figure 17 during evaluation.

In Section 5.3, we assume that each state was a single frame from the Atari 2600 simulator. [8] use states consisting of four past frames to provide the Q-function approximation neural network with additional context to better estimate Q-function values. To reduce the storage required for states, frames are preprocessed before being combined into states. First, the most recent two past frames are maxpooled to ensure that all elements of the environment are visible to the agent, as due to hardware limitations, in some games not every element of the environment is shown on screen every frame. Next, the frame is converted to greyscale. Finally, bilinear downsampling is applied to downsample the frame from a resolution of (210×160) to a resolution of (84×84).

To prevent overfitting since training and evaluation of the agent occurs in the same environment, the environment is placed in a random state during reset by performing no-operation (i.e. fire button unpressed and joystick centered) actions before the agent is allowed to act. During training, rewards are clipped between -1 and 1 to enable using a single set of hyperparameters for all games tested, while an episode (i.e. an Atari game session) terminates when a single life is lost to aid state value estimation.

As discussed in Section 7.3 below, the bottleneck in the algorithm of [8] is the speed of the Atari 2600 simulator, and hence to speed up training, [8] use frame-skipping, where the agent only observes and acts on every fourth frame and the action selected by the agent after its most recent observed frame is repeated during the three skipped frames following the agent's most recent observed frame, and perform a Q-function value approximation neural network optimization iteration every fourth frame the agent observes.

[8] anneals ϵ using a linear schedule and utilizes RMSprop [3] to optimize the Q-function approximation neural network.

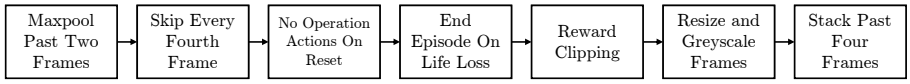

Figure 16: Order of Frame Preprocessing and Environment Modification Transformations During Training

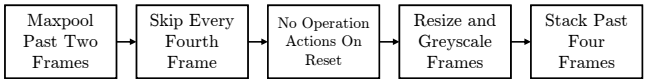

Figure 17: Order of Frame Preprocessing and Environment Modification Transformations During Evaluation

The hyperparameters used by [8] are shown below.

Hyperparameter	Value	Description
minibatch size	32	Number of training cases over which each stochastic gradient descent (SGD) update is computed.
replay memory size	1000000	SGD updates are sampled from this number of most recent frames.
agent history length	4	The number of most recent frames experienced by the agent that are given as input to the Q network.
target network update frequency	10000	The frequency (measured in the number of parameter updates) with which the target network is updated (this corresponds to the parameter *C* from Algorithm 1).
discount factor	0.99	Discount factor gamma used in the Q-learning update.
action repeat	4	Repeat each action selected by the agent this many times. Using a value of 4 results in the agent seeing only every 4th input frame.
update frequency	4	The number of actions selected by the agent between successive SGD updates. Using a value of 4 results in the agent selecting 4 actions between each pair of successive updates.
learning rate	0.00025	The learning rate used by RMSProp.
gradient momentum	0.95	Gradient momentum used by RMSProp.
squared gradient momentum	0.95	Squared gradient (denominator) momentum used by RMSProp.
min squared gradient	0.01	Constant added to the squared gradient in the denominator of the RMSProp update.
initial exploration	1	Initial value of ε in ε-greedy exploration.
final exploration	0.1	Final value of ε in ε-greedy exploration.
final exploration frame	1000000	The number of frames over which the initial value of ε is linearly annealed to its final value.
replay start size	50000	A uniform random policy is run for this number of frames before learning starts and the resulting experience is used to populate the replay memory.
no-op max	30	Maximum number of "do nothing" actions to be performed by the agent at the start of an episode.

Figure 18: List of Hyperparameters and their Values as Used in [8]

7.2.2 Q-Function Value Approximation Neural Network Architecture

The Q-function value approximation neural network used in [8] was illustrated previously in Figure 13. The exact architecture of the Q-function value approximation neural network is shown in the table below.

Layer	Type	Output Size	Kernel Size	Stride	Number of Parameters
Input	N/A	(`batch_size`, 4, 84, 84)	N/A	N/A	0
1	Convolution	(`batch_size`, 32, 20, 20)	8	4	8224
2	ReLU	(`batch_size`, 32, 20, 20)	N/A	N/A	0
3	Convolution	(`batch_size`, 64, 9, 9)	4	2	32832
4	ReLU	(`batch_size`, 64, 9, 9)	N/A	N/A	0
5	Convolution	(`batch_size`, 64, 7, 7)	3	1	36928
6	ReLU	(`batch_size`, 64, 7, 7)	N/A	N/A	0
7	Flatten	(`batch_size`, 3136)	N/A	N/A	0
8	Linear	(`batch_size`, 512)	N/A	N/A	1606144
9	ReLU	(`batch_size`, 512)	N/A	N/A	0
10	Linear	(`batch_size`, `num_actions`)	N/A	N/A	512 × `num_actions` + `num_actions`

Table 2: Architecture of the Q-Function Value Approximation Neural Network

Note that dimensionality reduction is done via strided convolution layers rather than via pooling layers. All layer options not mentioned use default PyTorch values.

7.3 Implementation

7.3.1 Overview

The overall code architecture is as shown below in Figure 19. We implemented the method of [8] in Python using OpenAI Gym [1] and PyTorch. While we used PyTorch to implement training and inference of the Q-function value approximation neural network, we utilized the Atari 2600 simulator as implemented by OpenAI Gym as the environment simulator. Each frame preprocessing and environment modification transformation was implemented using the `gym.Wrapper` API of OpenAI Gym to enable modularity and ease of testing different combinations of frame preprocessing and environment modification transformations. Using TensorboardX [4], we implemented a logging system to enable visualization of training progress.

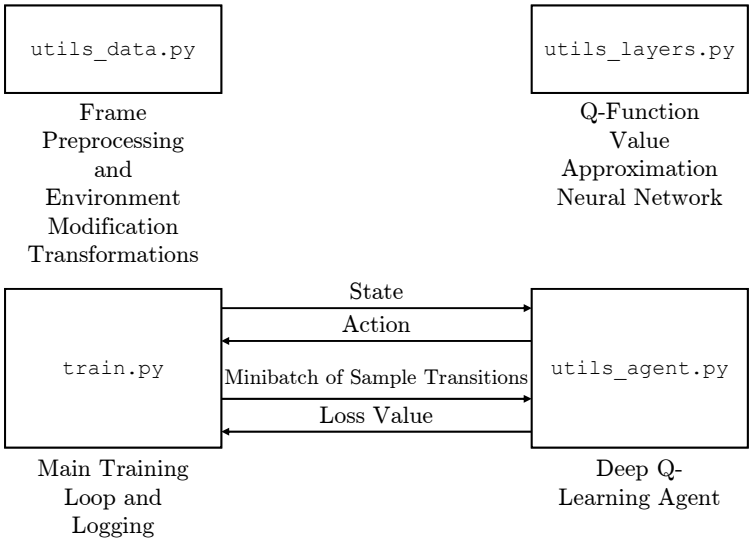

Figure 19: Code Architecture

Algorithm 6: The Algorithm of [8] As Implemented

Initialize the Atari 2600 simulator.
Initialize $\mathcal{D}$.
Initialize θ.
$\theta^{-} = \theta$
$\epsilon = \epsilon_{initial}$
total_frames = 0
previous_evaluation_frames = 0
while total_frames < number of frames to train **do**
    $\pi(s) = \text{argmax}_{a \in \mathcal{A}_s} Q_\theta(s, a)$
    Perform ϵ-greedy action and obtain a sample transition (s, a, r, s') from the environment and add (s, a, r, s') to $\mathcal{D}$
    if $\epsilon \geq \epsilon_{final}$ **then**
        Reduce the value of ϵ according to some schedule.
    end
    if Atari game session has terminated **then**
        Reset Atari 2600 simulator.
        total_episodes = total_episodes + 1
    end
    if $|\mathcal{D}| \geq$ minimum number of sample transitions to collection before starting training **then**
        Sample minibatch $\mathcal{B} \subseteq \mathcal{D}$
        Compute target

$$y_j = \left\{ \begin{array}{ll} r_j & \text{if } s_{j+1} \text{ corresponds to a terminal state} \\ r_j + \gamma \max_a Q_{\theta^-}(s_{j+1}, a) & \text{else} \end{array} \right\} \forall j \in \mathcal{B}$$

        Perform one iteration of SGD to minimize $\sum_{(s_j, a_j, r_j, s_{j+1}) \in \mathcal{B}} \left(Q_\theta(s_j, a_j) - y_j\right)^2$ w.r.t parameters θ.
    end
    if total_episodes - previous_evaluation_episodes $\geq$ environment_evaluation_interval_episodes **then**
        $\theta^{-} = \theta$
    end
    if total_frames - previous_evaluation_frames $\geq$ agent evaluation interval **then**
        Evaluate the agent using ϵ-greedy action selection with policy $\pi(s) = \text{argmax}_{a \in \mathcal{A}_s} Q_\theta(s, a) \, \forall s \in \mathcal{S}$ and $\epsilon = 0.05$
    end
    total_frames = total_frames + 1
end
return θ

7.3.2 Optimizations

In practice, because the method of [8] is synchronous (w.r.t. collecting sample transitions and gradient descent steps performed on the Q-function value approximation neural network) and the Q-function value approximation neural network has very few layers and parameters by contemporary standards, the method of [8] is typically bottlenecked by the speed of the Atari 2600 simulator used, and hence training speed is typically determined by CPU single-thread performance rather than GPU performance or GPU memory capacity provided frame preprocessing. As Q-Learning is

capable of learning from off-policy sample transitions, i.e. sample transitions that are not generated by an agent performing actions according to the current known best policy, a simple solution to speed up training is to use multiple simulators and multiple GPUs asynchronously [6].

Additionally, despite the graphical capabilities of Atari 2600, a replay buffer of one million sample transitions requires a significant amount of memory. As each state contains four past frames, creating a new array for each state results in significant wasted memory. Hence, when storing sample transitions in the replay buffer, we only store references to each frame, and create new arrays lazily as needed for input to the Q-function value approximation neural network. With this optimization, our implementation uses approximately 11GB of memory to store one million sample transitions, whereas an earlier implementation without this optimization would run out of memory on a computer with 32GB of memory.

7.3.3 Hyperparameters

We aimed to use the Adam optimizer [5] due to its faster convergence while keeping as many hyperparameters unchanged as possible. To do so, we decreased the value of ϵ_{final} to 0.01, doubled the number of sample transitions to store in the replay buffer before commencing Q-function value approximation neural network training, reduced the learning rate to 0.0001, decreased the number of actions to take between successive optimization iterations (update frequency) to 1, and decreased the number of frames between target network parameter updates to 1000. Additionally, we limit the number of actions by assuming that the joystick is four-directional and that the fire button cannot be pressed unless the joystick is centered.

[6] trained for 50000000 frames. Due to computational power limitations, we train for 10000000 frames and evaluate our final trained agent on 135000 frames during testing. All other hyperparameters were unchanged.

Hyperparameter	Value	Description
environment_train_frames_max	10000000	Total number of frames to train agent for.
environment_evaluation_episodes	10	Number of episodes to evaluate agent for per agent evaluation.
environment_evaluation_interval_episodes	500	Number of episodes between agent evaluations.
environment_checkpoint_interval_episodes	500	Number of episodes between agent checkpoints.
environment_no_op_max	30	Maximum number of no-operation (i.e. fire button unpressed and joystick centered) actions taken before agent is allowed to act.
environment_reward_clip_limit	1.0	Value to clip rewards between.
environment_frames_to_maxpool	2	Number of past frames to maxpool.
environment_state_height	84	Height of frames input to Q-function value approximation neural network.
environment_state_width	84	Width of frames input to Q-function value approximation neural network.
agent_history_length	4	Number of past frames in a single state input to Q-function value approximation neural network.
agent_action_repeat	4	Number of frames to wait between successive agent actions.
agent_update_frequency	1	Number of actions to take between successive optimization iterations.
agent_discount_factor	0.99	Value of the discount factor γ.
agent_epsilon_greedy_epsilon_initial	1.0	Value of $\epsilon_{initial}$.
agent_epsilon_greedy_epsilon_final	0.01	Value of ϵ_{final}
agent_epsilon_greedy_annealing	1000000	Number of frames to anneal ϵ over.
agent_epsilon_greedy_epsilon_evaluation	0.05	Value of ϵ during agent evaluation.
agent_replay_buffer_initial	100000	How many sample transitions to store in the replay buffer before commencing Q-function value approximation neural network training.
agent_replay_buffer_maximum_length	1000000	Maximum number of sample transitions to store in the replay buffer.
optimizer_name	Adam	Optimization algorithm used.
optimizer_minibatch_size	32	Minibatch size used during optimization.
optimizer_q_network_parameter_update_interval_frames	1000	Number of frames between target network parameter updates.
optimizer_learning_rate	0.0001	Learning rate used during optimization.

Table 3: List of Hyperparameters and their Values as Used in Our Implementation

7.4 Analysis of the Agent's Performance on Each Game Tested

7.4.1 Breakout

The agent successfully learns the optimal strategy, which is to dig a tunnel past the bricks and then bounce the ball through the tunnel so that the ball will keep bouncing back and forth between the bricks and the back wall.

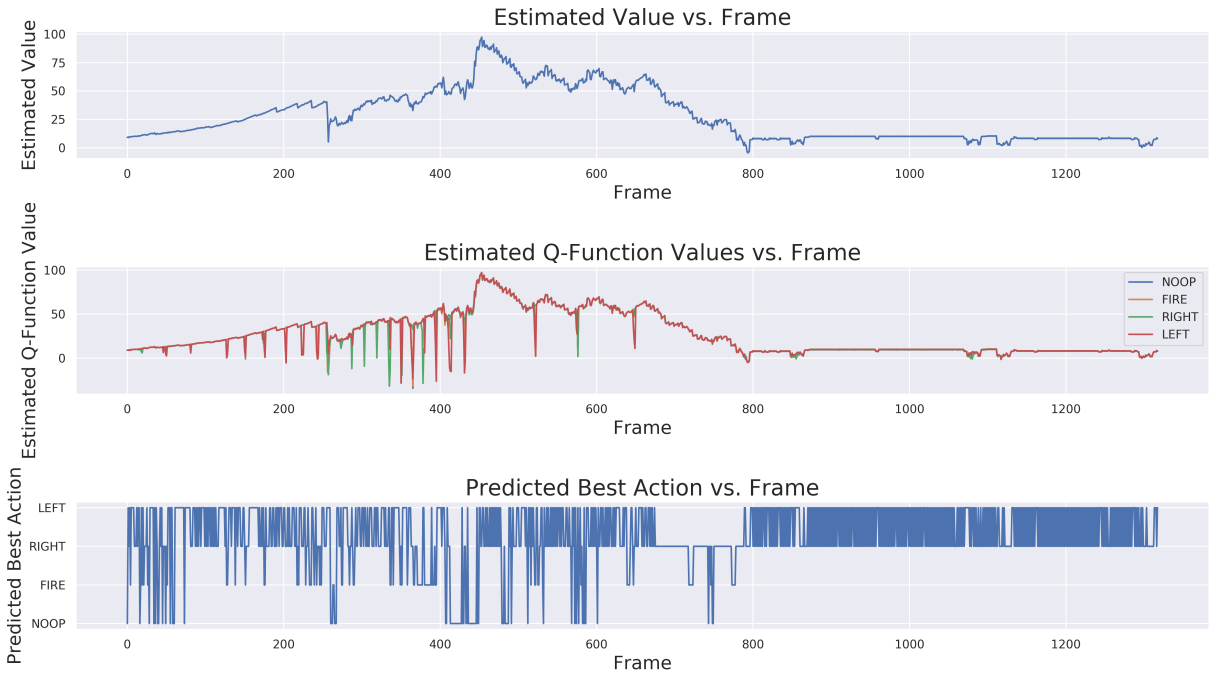

Figure 20: Agent Estimated Value, Estimated Q-Function Value, and Predicted Best Action vs. Frame

In this evaluation episode, the agent achieved a score of 361 points. At around frame 259, the agent almost fails to catch the ball and hence the estimated value drops abruptly.

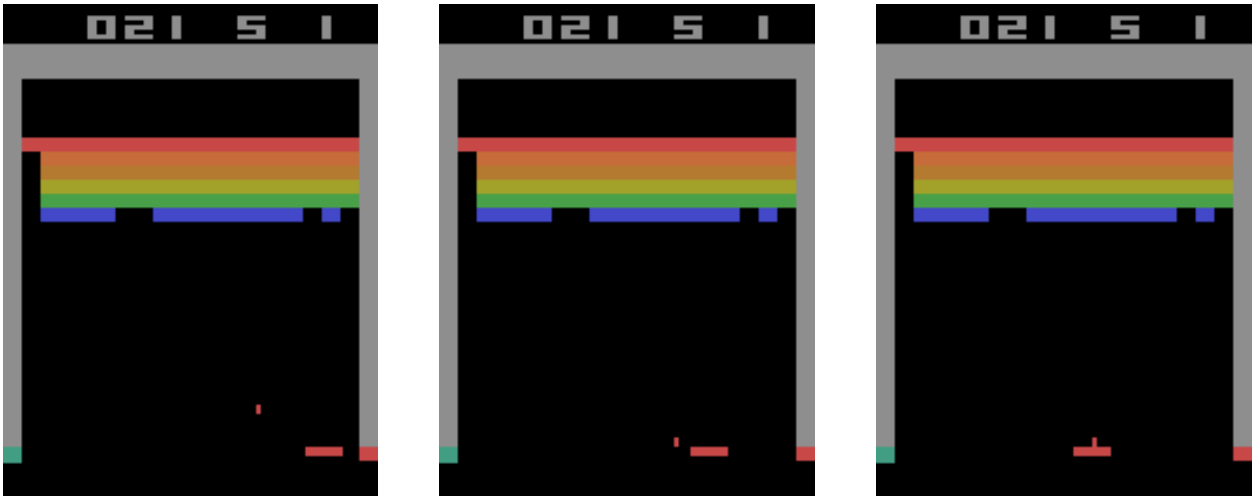

Figure 21: Frame 258, Frame 259, and Frame 260

At around frame 440, the agent has successfully dug a tunnel through the bricks and bounced the ball through the tunnel, so the estimated value rises dramatically.

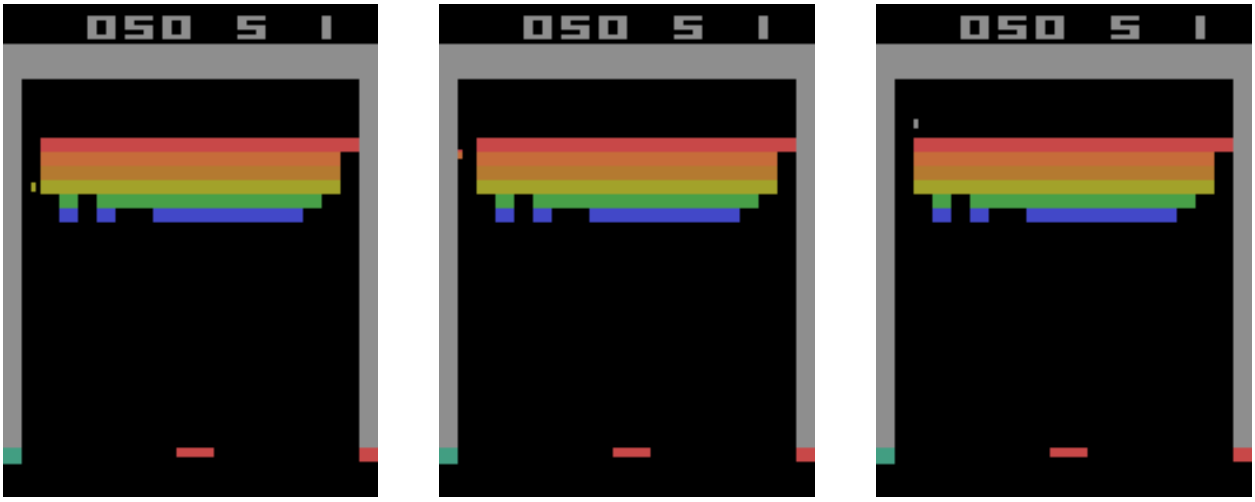

Figure 22: Frame 441, Frame 442, and Frame 443

7.4.2 Enduro

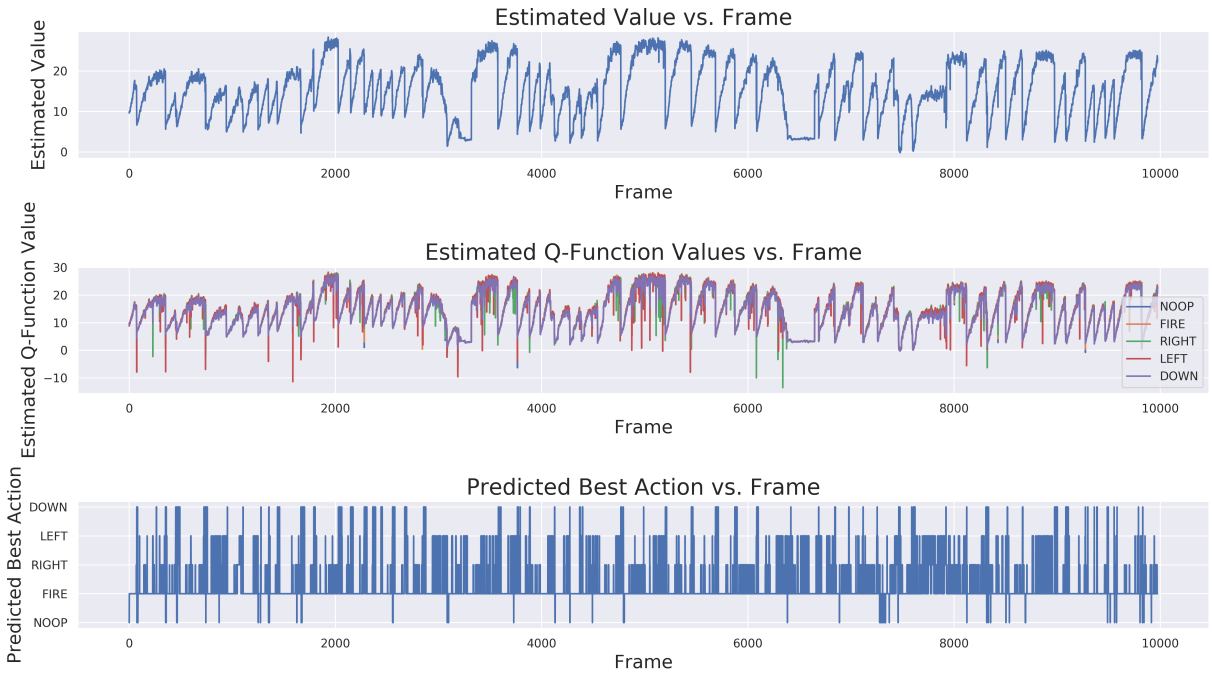

Figure 23: Agent Estimated Value, Estimated Q-Function Value, and Predicted Best Action vs. Frame

In this evaluation episode, the agent achieved a score of 634 points.

Figure 24: Frame 6833, Frame 6834, and Frame 6835

7.4.3 Pong

The agent successfully learns the optimal strategy, which is to bounce the ball toward the opposing player at a specific speed and angle. This necessitates that the agent hit the ball using the edge of its paddle.

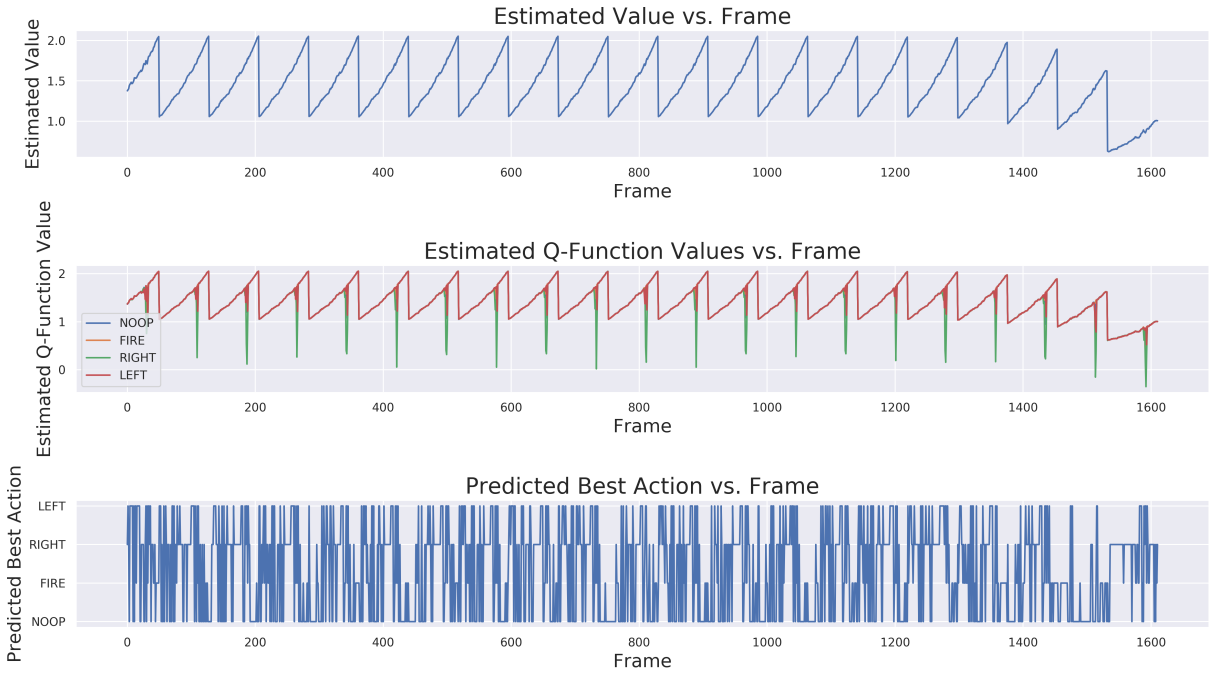

Figure 25: Agent Estimated Value, Estimated Q-Function Value, and Predicted Best Action vs. Frame

In this evaluation episode, the agent achieved a score of 21 points. At around frame 204, the ball goes past the paddle of the opposing player and the agent gains 1 point. Hence, the estimated value drops abruptly.

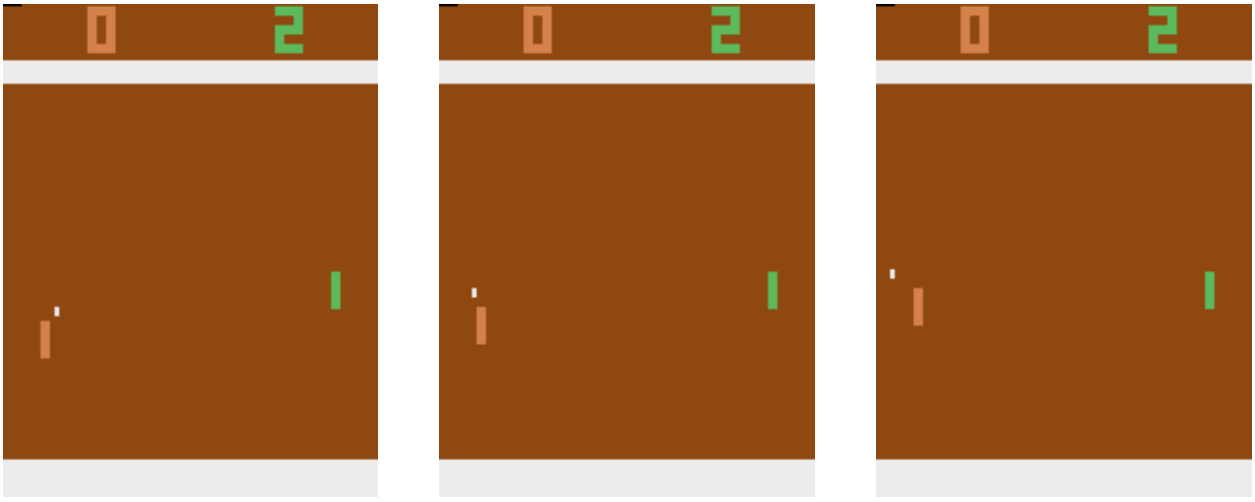

Figure 26: Frame 203, Frame 204, and Frame 205

We see below that the agent does indeed hit the ball using the edge of its paddle.

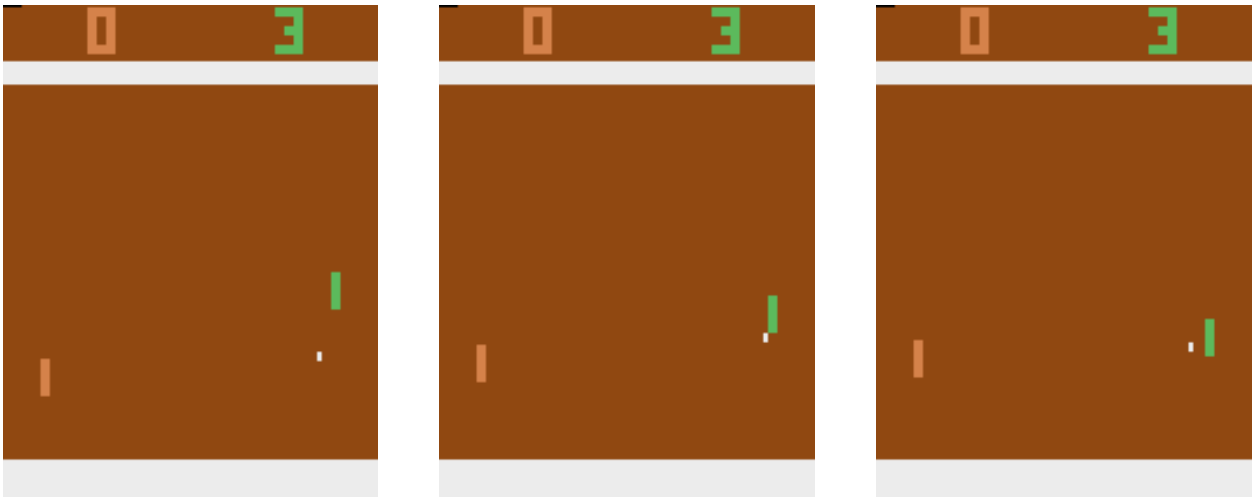

Figure 27: Frame 265, Frame 266, and Frame 267

7.4.4 River Raid

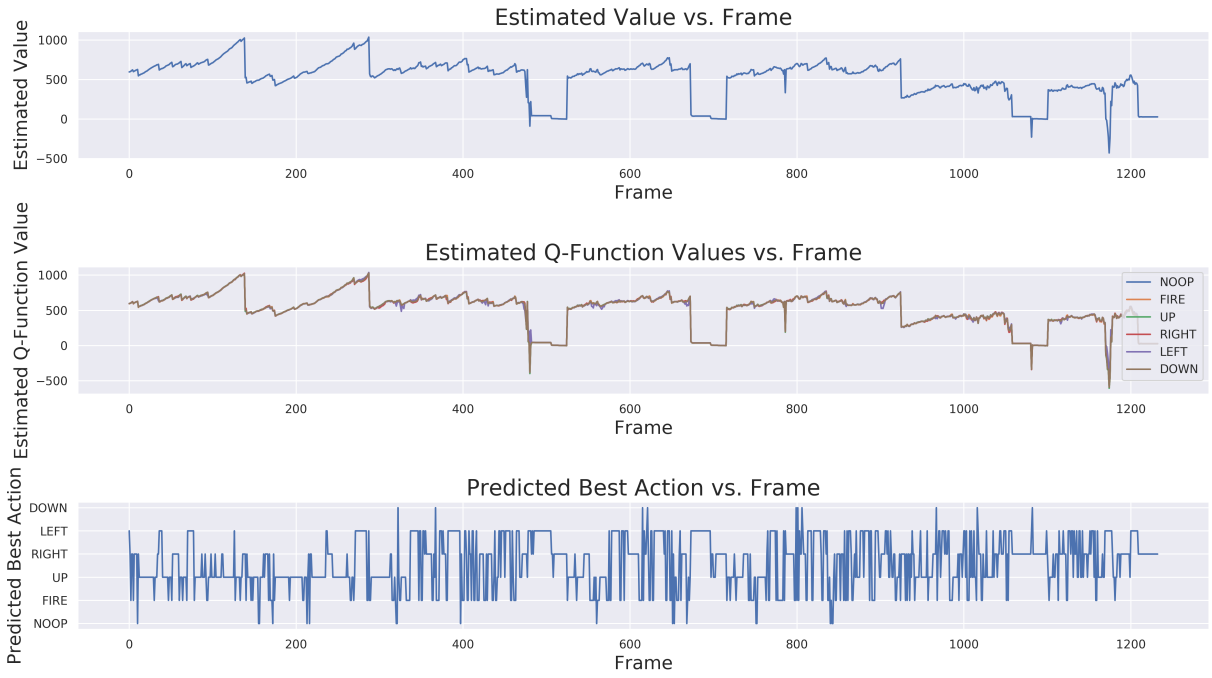

Figure 28: Agent Estimated Value, Estimated Q-Function Value, and Predicted Best Action vs. Frame

In this evaluation episode, the agent achieved a score of 5430 points. At around frame 140, a barrier that must be destroyed in order to proceed is destroyed by the agent. Hence, the estimated value rises and drops abruptly.

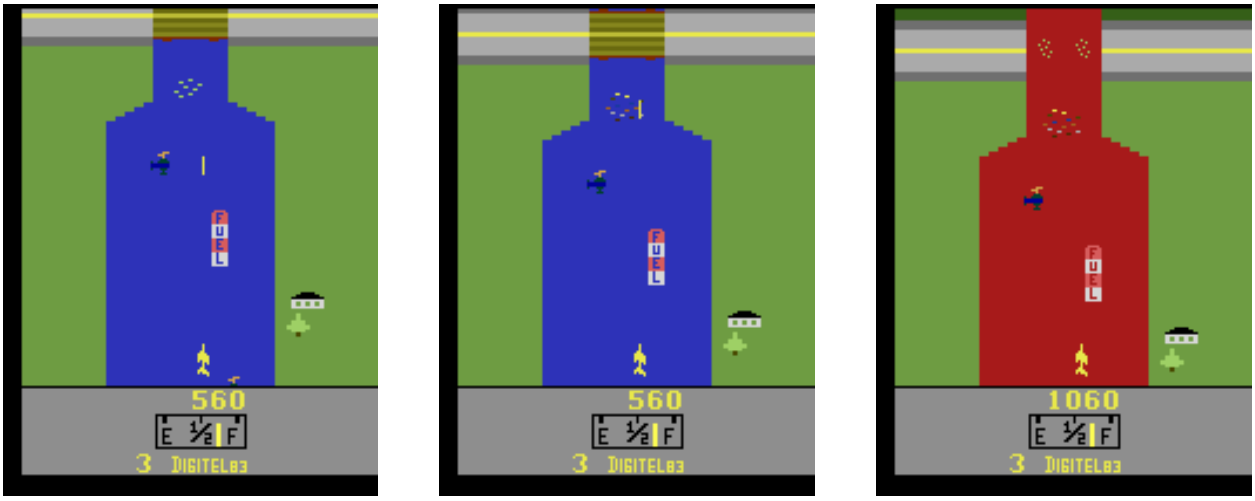

Figure 29: Frame 137, Frame 138, and Frame 139

At around frame 480, the aircraft controlled by the agent is about to collide with a ship. Hence, the estimated value drops abruptly.

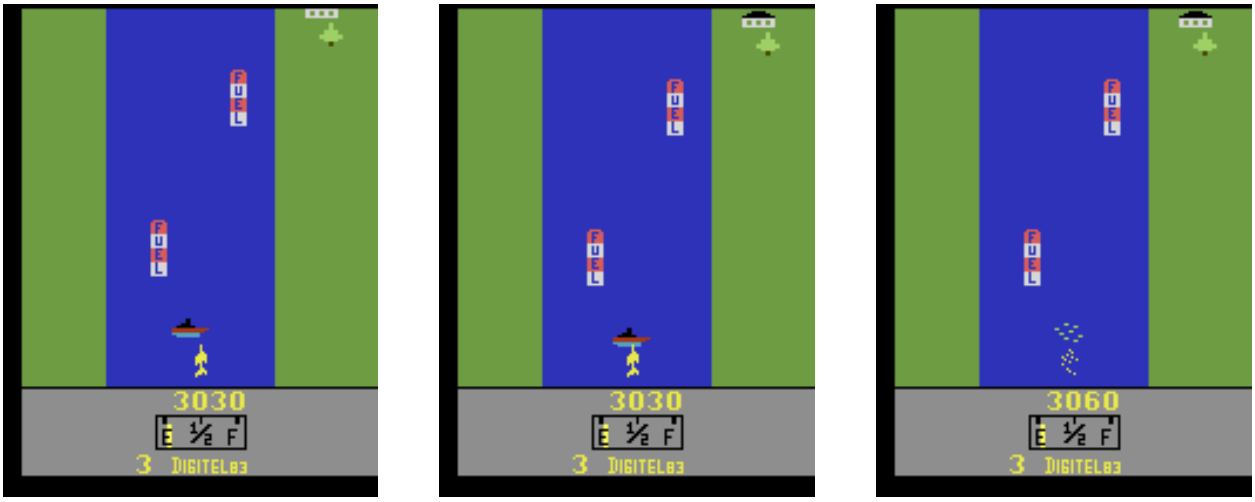

Figure 30: Frame 480, Frame 481, and Frame 482

7.4.5 Space Invaders

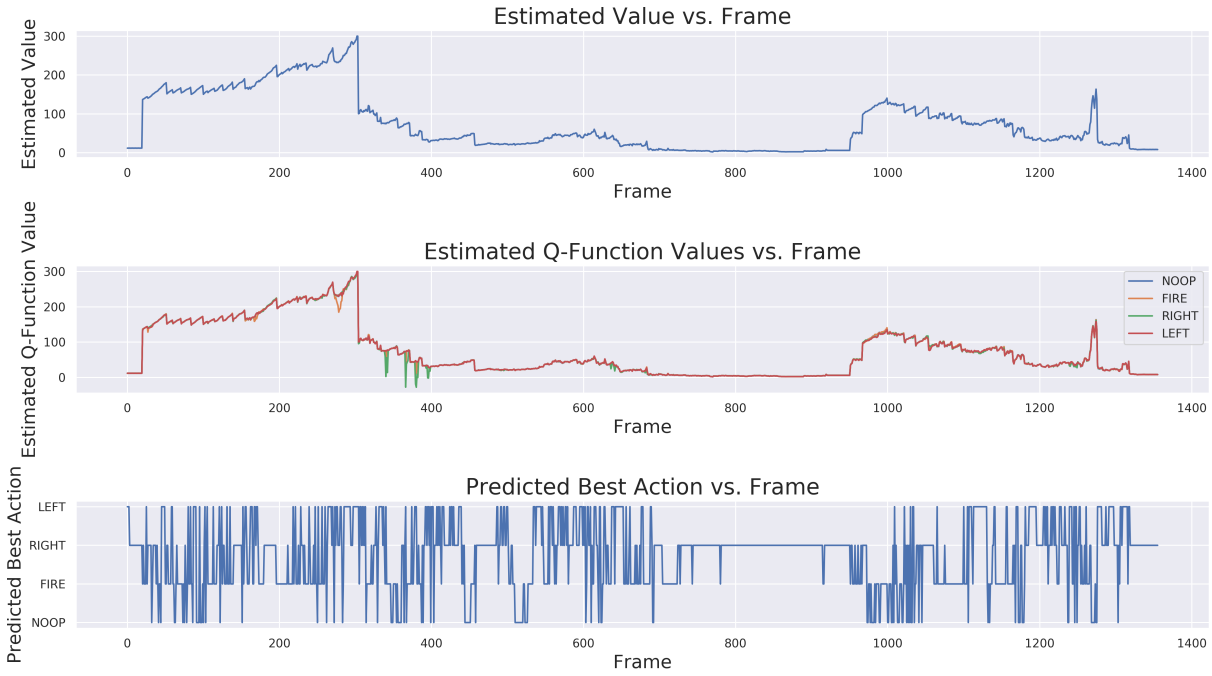

Figure 31: Agent Estimated Value, Estimated Q-Function Value, and Predicted Best Action vs. Frame

In this evaluation episode, the agent achieved a score of 1345 points. Note that at around frame 304, the agent prioritizes destroying the Alien Mothership as destroying the Alien Mothership gives a much greater reward as compared to that of destorying regular alien ships.

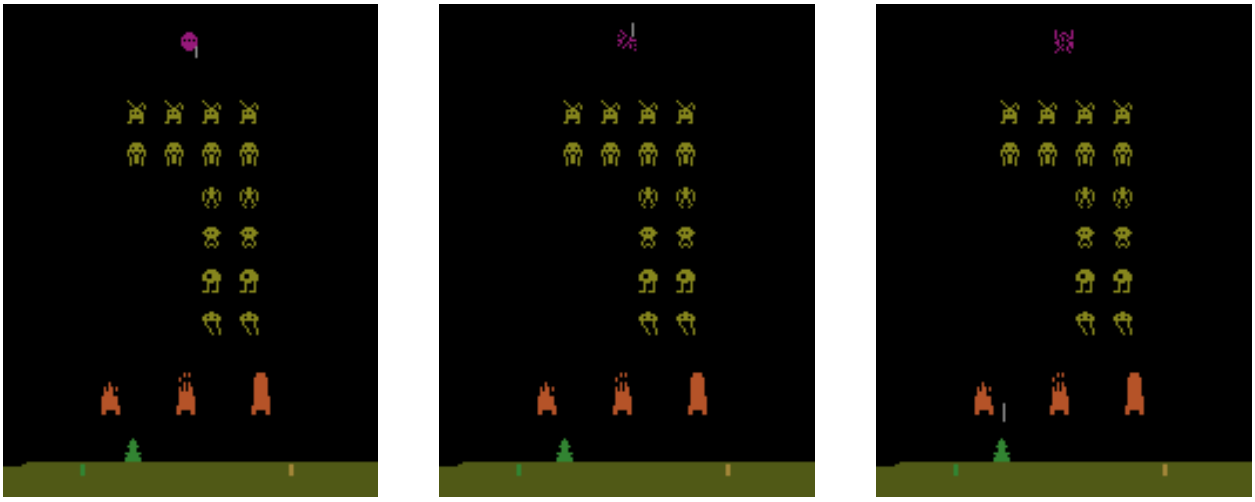

Figure 32: Frame 304, Frame 305, and Frame 306

7.5 Experiments and Results

7.5.1 Q-Function Value Approximation Neural Network Initialization Strategy

We compare three different convolution layer weight initialization methods.

Initialization Method	Episode Score Mean	Episode Score Variance	Episode Frame Length Mean	Episode Frame Length Variance
PyTorch Default	11.174	696.84	480.43	22894
He [2]	22.865	749.63	703.13	87811
Orthogonal [9]	32.154	3695.1	576.92	29401

Table 4: Q-Function Value Approximation Neural Network Initialization Strategy Study on Breakout

We hence use orthogonal initialization for all subsequent experiments.

7.5.2 Game Performance Comparison

Game	Episode Score Mean	Episode Score Variance	Episode Frame Length Mean	Episode Frame Length Variance
Breakout	32.154	3695.1	576.92	29401
Enduro	258.63	14905	4500.0	2782600
Pong	19.083	5.4014	1875.0	56022
River Raid	3459.4	877420	833.33	37002
Space Invaders	748.14	30417	882.35	41330

Table 5: Game Performance Comparison

7.5.3 Ablative Study

To study the effect of the target network and the replay buffer, we train an agent on Breakout and disable the target network and/or the replay buffer. The results demonstrate that both modifications contribute to the efficacy of Deep Q-Learning.

Target Network	Replay Buffer	Episode Score Mean	Episode Score Variance	Episode Frame Length Mean	Episode Frame Length Variance
Yes	Yes	32.154	3695.1	576.92	29401
No	Yes	10.549	1.4201	639.81	27774
Yes	No	7.4837	3.9444	438.31	35112
No	No	10.701	1.2104	671.64	38953

Table 6: Ablative Study on Breakout

7.6 Conclusion

Experimental results demonstrate that our implementation results in agents with comparable performance to that of [8] on Breakout and Pong. More hyperparameter tuning is required to enable our implementation to match the performance of [8] on other Atari games.

References

[1] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *CoRR*, abs/1606.01540, 2016.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, Santiago, Chile, December 2015. IEEE.

[3] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. rmsprop: Divide the Gradient by a Running Average of its Recent Magnitude, 2014.

[4] Tzu-Wei Huang. lanpa/tensorboardX, December 2019. original-date: 2017-06-13T13:54:19Z.

[5] Diederik P. Kingma and Jimmy Lei Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, San Diego, CA, USA, 2015.

[6] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, June 2016. arXiv: 1602.01783.

[7] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing Atari with Deep Reinforcement Learning. *CoRR*, abs/1312.5602, 2013.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[9] Andrew M. Saxe, James L. McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv:1312.6120 [cond-mat, q-bio, stat]*, February 2014. arXiv: 1312.6120.