Expediting Learning with Interactive Demonstrations

Brandon Rozek

University of Mary Washington

Abstract

Deep Reinforcement Learning has shown great progress in domains such as the Atari Arcade Learning Environment. The problem, however, is that the agent playing the game requires many interactions before it starts to show good performance. This is okay for some domains such as video games, but as we start to look towards integrating deep reinforcement learning into real world applications, we need to minimize the number of interactions required. Interactive demonstrations help expedite the agent's learning process by providing a shared environment between the agent and the demonstrator to take turns in. This helps the agent learn directly from the demonstrator and allows the demonstrator to correct deviations that the agent made from the task. This approach is more natural to implement than other similar imitation learning techniques and has shown to reach a better level of performance faster than an enhanced Deep Q-Learning Network (DQN).

Contents

Introduction

Deep reinforcement learning has shown great progress in complex sequential decision making problems. Interest in this field revitalized thanks to Mnih et al. (2013) as they had great success designing a single reinforcement learning algorithm to play seven Atari 2600 games. These games feature simplistic 2D graphics and low-strategy game elements. At first, this may seem like a simple feat. However, humans have prior knowledge of how games work. A reinforcement learning agent starts learning to play the game from *tabula rasa* or blank slate. Consequently, the agent has to learn even simple features like what different shapes represent in a game. Since Atari 2600 games provide an appropriate balance of difficulty, they are often used as the benchmark for deep reinforcement learning algorithms. Bellemare et al. (2013) provided an easy to use library for other researchers to benchmark their agents called the Arcade Learning Environment (ALE). This allows researchers to standardize and easily compare the results of different approaches.

Since reinforcement learning agents are learning from tabula rasa, they often require hundreds if not thousands of interactions with an environment to learn a task. This is possible in the domain of video games as the agent can play a game thousands of times with no consequence. However, not all domains can offer this luxury. For example, many real world problems do not come with an accurate simulator. Even if it were possible to create one, some organizations do not want to spend the resources to do so. Instead, the agent must learn the task with actual consequences for its actions. In robotics, this can result in physical damage to the agent or objects in the environment. This creates a need for *sample-efficient* reinforcement learning algorithms. The term sample-efficient means that an agent learns from fewer samples or interactions with an environment.

Even though simulators don't exist for all domains, in most cases, it is easier to have access to demonstrations of the task. Learning from demonstration data can help expedite learning but comes with its own challenges. First of all, demonstrations often do not cover all the edge cases within the environment. This is similar to "If nothing goes wrong, here are the actions to perform." Due to this, agents often can not copy the demonstrator's actions verbatim as most environments are stochastic in nature or have a random component. This often leads to compounding errors as highlighted by Figure 1 since the agent deviates each time it fails to copy the demonstrator.

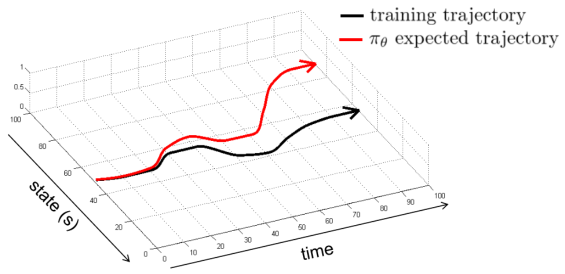

Figure 1: Deviations in Trajectory by Levine (2017)

As the agent and the demonstrator start to deviate significantly, it becomes difficult for the agent to learn from the demonstrator. Deviations are difficult to correct due to the high dimensional nature of many environments. For example in the ALE, it would be hard for the agent if not impossible to recreate the same exact pixel screen as before it deviated. All of this makes learning from static demonstrations difficult. Interactive demonstrations, however, combat this by having an agent and demonstrator take turns interacting within a shared environment. Therefore, the agent can interact freely with the environment and the demonstrator can then correct deviations in order to complete the task. This in turn, helps the agent learn how to succeed under many different scenarios that it would not have gotten to experience otherwise.

Background

Definitions

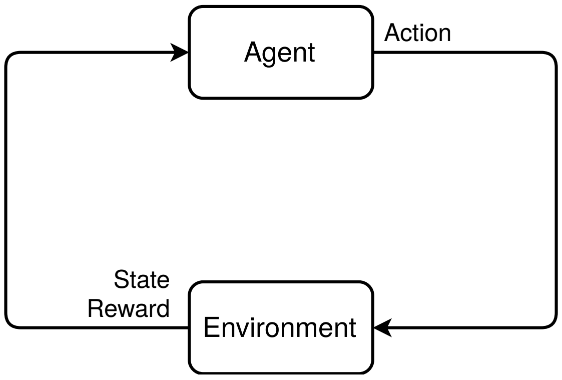

Figure 2: Agent-Environment Workflow

Reinforcement Learning addresses the problem of an agent learning how to act within an environment in order to maximize some reward signal. The agent interacts with the environment through a series of sequential decisions. As shown by Figure 2, the environment responds to the actions the agent takes by presenting states and rewards. A state encodes information relevant to the task at hand and a reward indicates the agent's progress on that task. The agent uses the states provided by the environment to determine from a list of possible actions the next action it will take. If the state does not contain all the information an agent needs to make the best decision, we call the state an image. When training the agent, it is important to ensure that it has as much information as possible in order to make the best decision. We will describe different preprocessing techniques in the Environment section to help the agent have the information it needs while cutting out irrelevant details.

One way to formalize this interaction is via a Markov Decision Process (MDP). States in a MDP are required to be *markovian* which means that future states are only dependent upon the current one. This is why we want our states to contain enough relevant information for the task.

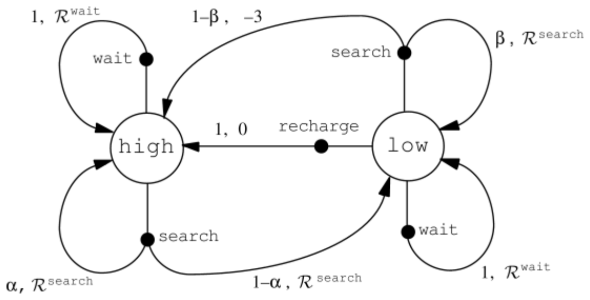
1, $\mathcal{R}^{\text{wait}}$
wait
1−β, −3
search
β, $\mathcal{R}^{\text{search}}$
1, 0
recharge
high
low
search
wait
α, $\mathcal{R}^{\text{search}}$
1−α, $\mathcal{R}^{\text{search}}$
1, $\mathcal{R}^{\text{wait}}$

Figure 3: MDP of Recycling Robot Example (Sutton and Barto 2018)

In the Recycling Robot Example in Figure 3 (Sutton and Barto 2018) we have two states `high` and `low`. When the robot is in the `high` state it can perform two actions: `wait` and `search`. Meanwhile, when the robot is in the `low` state it can perform three actions: `wait`, `search`, and `recharge`. The edges of the arrows are annotated with two values p, and R, where p is the probability of that state and action resulting in the new state and R represents the reward received from the environment during that transition. In this paper we are going to focus on a model free approach. This means that we do not know the state transition probabilities ahead of time.

The agent's goal is to learn a policy, which is a mapping from states to actions, that maximizes some reward function. The reward function needs to be crafted in order to incentivize the agent to perform the task given and nothing else. Otherwise, the agent might attempt to solve a task in an unexpected or sub-optimal way. An example from real life is in education. In this case, the students are agents whose reward function is the scores received on tests. Some students maximize their score by cheating off another student. In that way, the student receives a high score without achieving the original goal intended.

Due to this, most researchers design their reward functions to be sparse. Sparse means that for the majority of agent-environment interactions the reward is zero. With this approach the reward is nonzero depending on whether or not the agent fails or succeeds at the task in the end. For example, it may feel natural to assign points for capturing pieces in a game of chess. However, the agent might sacrifice a win in order to capture its opponent's pieces. Therefore in chess, it is common to assign a $+1$ for a win and a -1 for a loss at the end. Receiving sparse rewards makes learning more challenging for the agent as it does not receive feedback for its actions until the end of an interaction. This furthers the need for a demonstrator to provide an additional feedback signal to the agent during training.

Differences to Other Approaches

Reinforcement learning is different from other methods of machine learning. For example in supervised learning, the agent would be given the best actions for some states and its goal would be to predict the best actions from states it has not seen. Recall, however, that in reinforcement learning we do not know what the best actions for each state are. Instead, a reward function is defined for a given task, and the goal is to find actions for each state that maximize said reward function.

We do not attempt to solve the reinforcement learning problem in general. Instead we focus our approach by making a few assumptions about the problem. First, we concern ourselves with episodic tasks which means that there are a finite number of interactions that an agent can have with the environment. Next, we focus on environments with discrete action spaces. This means that an agent selects an action from a list of possible actions as opposed to providing continuous values. An example of continuous actions occur when driving a car. The agent would need to apply a certain amount of force to the pedal in order for the car to accelerate. Also, the agent would have to turn the wheel a certain number of degrees in order to steer the car. Finally as described earlier, the techniques in this paper are model-free. This means that we do not attempt to learn the state dynamics of the environment. In other words, the agent does not attempt to predict what will happen after it performs an action.

Components of Reinforcement Learning Algorithms

When it comes to learning, there are a few components that reappear across different approaches. First of all, learning typically occurs when the agent looks back at its past experiences. In order to do this, there needs to be some sort of memory module in place that allows the agent to sample past memories. Then, the agent needs a way to approximate some property of the MDP. In this paper, we'll focus on Value-based methods which mean that we're trying to approximate the value or the expected return of rewards from a given state. Throughout training we will improve this approximation by using the past experiences gathered by the agent. This is done by comparing approximations produced to what we've seen in the past. By doing this, we receive a loss/error which can then be used to further improve the approximation. The approximation is then used to derive a policy. Together, the memory of past experiences, approximation of the MDP property, the error in estimation, and the policy derived create a reinforcement learning algorithm. This is shown graphically in Figure 4. For the rest of the Background section we will go over each component in depth.

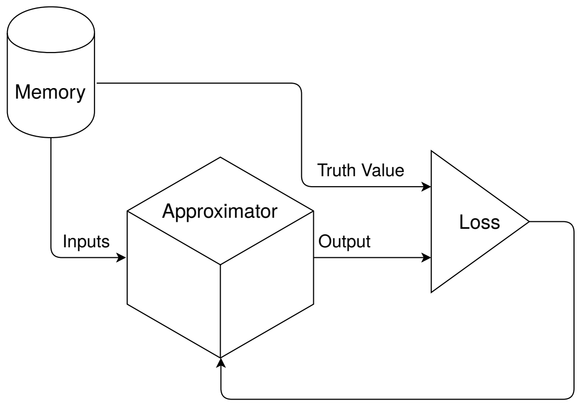

Figure 4: Improving the Approximator

Memory

As mentioned before, we need a way to store past memories into a buffer D. Let us define t to be the current time-step of the agent within the environment. Referring back to the MDP, a decision needs to be made by the agent before it transitions to a different (or possibly same) state. Therefore for the sake of consistency, a lack of action (called a NOOP) is also an action. At each time-step t, the agent at its current state will perform an action, receive a reward from the environment, and then transition into the next state. Thus, we define an experience e_t as a tuple containing the current state (s_t), the action taken (a_t), the reward received (r_t), and the state that the agent was transitioned to (s_{t+1}). In practice, storing these experiences can be memory intensive. Therefore, a max capacity $D_{capacity}$ is often set so that we only consider the last $D_{capacity}$ number of experiences.

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

After some interactions with the environment, the agent will look back at the experiences from D in order to train the approximator. It would be computationally inefficient to look back at all the experiences gathered, so only a small subset would be considered. A subset of experiences is called a *batch*. Typically the number of experiences sampled is constant throughout training and is represented as $batch_{size}$.

It is important to not just look at a single experience when calculating the loss of an approximator, as that might cause the adjustments to overfit specifically to that experience. The goal of the approximator is to represent some property of the overall MDP and not just the experiences received. This is especially important in environments where the possible state space is large, and therefore unlikely for an agent to experience it all during the training process.

Additionally, in order to satisfy conditions existent in many optimization techniques, the experiences in the batch must be independent and identically distributed (i.i.d.) This means that we can not sample a continuous segment in D as they are often highly correlated with one another. For example, let's sample five minutes from a typical day. If we were to sample five minutes when eating, each second would be highly similar to each other as opposed to if we took five minutes worth of seconds scattered throughout the day. Lin (1992) describes the memory buffer above, and discusses performing random sampling from D in order for the batch to be i.i.d. Figure 5 shows an example of a batch with size 4.

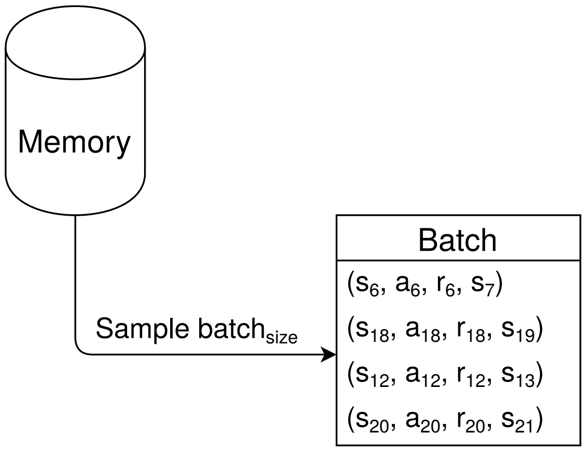
Memory
Sample batch$_{size}$
Batch
(s$_6$, a$_6$, r$_6$, s$_7$)
(s$_{18}$, a$_{18}$, r$_{18}$, s$_{19}$)
(s$_{12}$, a$_{12}$, r$_{12}$, s$_{13}$)
(s$_{20}$, a$_{20}$, r$_{20}$, s$_{21}$)

Figure 5: Example batch of size 4

Value Function

The goal of reinforcement learning algorithms is to produce a policy that maximizes the reward function R. One way we can maximize this function is to maximize a similar surrogate function. The value function (shown in Equation 1) is the expected discounted reward following a policy π from a certain state-action pair to the end of the interaction at time T. Intuitively, this function attempts to predict the outcome of performing a certain action at a given state.

$$Q(s_t, a_t) = \mathbb{E}_\pi[\sum_{t'=t}^{T}(\gamma^{t'}R(s_{t'}, a_{t'}))] \tag{1}$$

The amount that the reward is discounted is controlled by a fixed parameter γ. The discount factor controls how myopic the agent is. As $\gamma \to 1$, the agent weighs the rewards from future states the same as the current state. Meanwhile as $\gamma \to 0$, the agent tends to prefer shorter term gains. Reward estimation typically gets worse the farther into the future one tries to predict, therefore rewards far off in the future should be taken less into consideration when making decisions for the current moment. Thus, it is beneficial for $\gamma < 1$.

Loss Function

Loss functions are important to obtaining accurate approximators as they define an error in the approximator. Minimizing the loss function will then improve the approximator. An important property of the value function that will be beneficial for defining a loss function is called the Bellman Equation (shown in Equation 2). This states that the value from a given state-action pair can be found by calculating the reward from that pair and combining it with the result of the next value.

$$Q(s_t, a_t) = R(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}) \tag{2}$$

The loss can then be derived by what is called the Bellman update. This consists of comparing the expected value of the difference between the left hand side and the right hand side of the bellman equation. In literature, this loss is called the temporal difference (TD) loss (as shown in Equation 3). In a perfect approximation, this loss would be zero. However, this is usually not the case.

$$loss_{TD} = \mathbb{E}[Q(s_t, a_t) - (R(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}))] \tag{3}$$

We compute this in practice by sampling our batch of experiences $[e_i, \ldots, e_j]$ where i, j are time steps between 0 and the current time step t, computing individual losses, and combining them with the mean-squared function.

Computing individual loss:

$$loss_k = Q(s_k, a_k) - (r_k + \gamma \mathbb{E}_{a \sim \pi} Q(s_{k+1}, a))$$

where $(s_k, a_k, r_k, s_{k+1}) \in e_k$.

Aggregating individual losses:

$$loss = MS_{k \in batch}(loss_k) = \frac{\sum_{k \in batch} loss_k}{batch_{size}}$$

An optimization routine is then executed to minimize the loss.

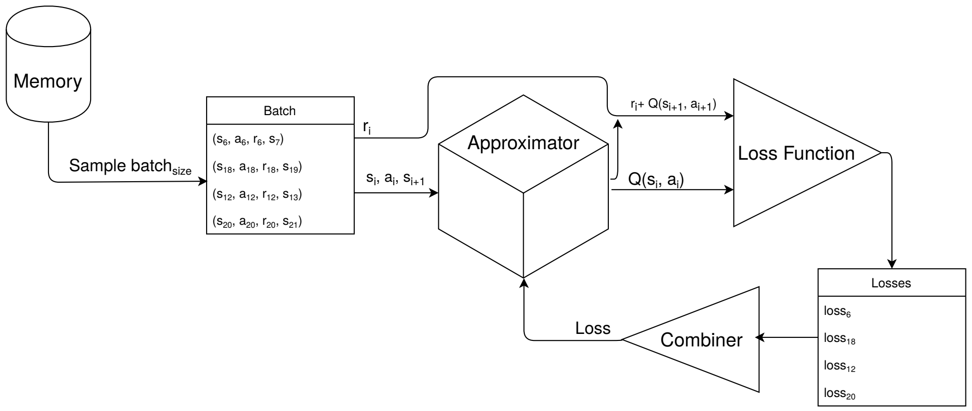

Figure 6: Loss Routine Example

Policy

We can now take the approximation of the value function and derive a policy. Since the goal of the reinforcement learning problem is to maximize rewards, the most obvious policy is to choose actions that maximize the value. This is otherwise called *greedy selection* as shown in Equation 4.

$$\pi(s) = argmax_{a \in A} Q(s, a) \tag{4}$$

where A is the possible actions for the given state s.

During training when the approximator is inaccurate, it is common not to use the greedy selection policy as it inhibits exploration. In order to avoid sub-optimal solutions, it is often common to have to perform what may seem like sub-optimal actions at the moment in order to maximize future rewards. An example of this is in the game Seaquest. In this game, as shown in Figure 7, the agent is a submarine who receives rewards for rescuing divers and shooting the fish. The game has an oxygen tank for the submarine that when empty ends the game. However, going to fill up the oxygen tank provides no rewards. If greedy selection was used during training, the agent might not ever discover that the game can continue past one oxygen tank.

Figure 7: Seaquest

Other common policies include epsilon-greedy and the softmax Bellman operator. Epsilon-greedy (Watkins 1989) is the most prevalent action selector chosen for training. It is the same as greedy selection, except that at a fixed or varying with time probability ϵ, a random action is chosen instead. The softmax Bellman operator (Bridle 1990) shown in Equation 5 takes the softmax or normalized exponential function of the values to produce a probability for taking each action.

$$\pi(s) = \sigma_{a \in A}(Q(s,a)) = \frac{e^{Q(s,a')}}{\sum_{a \in A} Q(s,a)} \tag{5}$$

Example Implementations

Deep Q-Networks Mnih et al. (2013) brought to light the effectiveness of deep reinforcement learning in the ALE. Deep reinforcement learning is the use of deep neural networks as the approximator in the learning algorithm. The approach outlined by Mnih et al. (2013) (shown in Algorithm 1) is also classified as an *off-policy* method, since the current policy is not used during the training process. During the loss calculation, the maximum possible Q-value for the next state is used instead.

$$loss_k = Q(s_k, a_k) - (r_k + \gamma max_{a \in A} Q(s_{k+1}, a))$$

where $(s_k, a_k, r_k, s_{k+1}) \in e_k$.

One important implementation detail for processing image input, is the usage of convolutional neural networks (CNNs). It is impractical to derive Q-values on a pixel by pixel basis, therefore CNNs take advantage of spatially local correlations (pixels nearby are similar) to minimize the dimensionality of the input. This reduction in dimensionality further helps narrow down what information is truly important to the agent and speeds up the learning process.

Improvements to Deep Q-Networks

Algorithm 1 Deep Q-Learning with Experience Replay

```
 1: Initialize replay memory $D$ to capacity $D_{capacity}$
 2: Initialize action-value function $Q$ with random weights $\theta$
 3: for episode $\in \{1, \dots, M\}$ do
 4:     Observe initial state $s_1$
 5:     Preprocess $s_1$ through $\phi$ to get $\phi_1$
 6:     for $t \in \{1, \dots, T\}$ do
 7:         With probability $\epsilon$ select a random action $a_t$
 8:         otherwise select $a_t = max_a Q^*(\phi(s_t), a; \theta)$
 9:         Execute action $a_t$ in the environment and observe reward $r_t$ and image $s_{t+1}$
10:         Preprocess $\phi_{t+1} = \phi(s_{t+1})$
11:         Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$, replacing the oldest experience if $|D| \geq D_{capacity}$
12:         Sample random batch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $D$
13:         Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
14:         Calculate $loss_j = y_j - Q(\phi_j, a_j; \theta)$
15:         Combine losses with mean-squared function to obtain $loss$
16:         Use $loss$ to perform a gradient descent step on the Q-estimator
```

Double Deep Q-Networks The Deep Q-Learning algorithm uses the same approximate value function to calculate the action that produces the highest value and the value themselves. Hasselt, Guez, and Silver (2016) discovered that this makes the policy more likely to select actions with overestimated values. This often leads to overoptimistic value estimates. They also note that overestimation is different than the exploration technique *optimism in the face of uncertainty* due to over-estimations not being uniform across actions and not being concentrated at states we wish to learn more about. To prevent this phenomenon, the selection of the action is decoupled from the calculation of the value by introducing what is called the target network.

The target network is architecturally the same as the regular Q-Network but its weights θ are updated less rapidly than in the regular online network. How the target network weights get updated varies by implementation. Some update the weights after a fixed number of iterations τ, others use a technique called Polyak-Ruppert Averaging by Polyak (1990) and Ruppert (1998). This technique nudges the weights of the target network towards the online network at every iteration as described by Equation 6.

$$\hat{\theta}^{(t)} = \alpha\hat{\theta}^{(t-1)} + (1-\alpha)\theta^{(t)} \text{ (where } 0 \leq \alpha \leq 1) \tag{6}$$

With this separation of online and target networks, the policy gets evaluated using the online network while the target network gets used to estimate its value. This technique as outlined in Algorithm 2 has been shown to lower over-estimations as well as lead to better performance.

Dueling Network Architecture Rewards obtained are often dependent upon the state itself. Therefore, instead of looking at each Q-value independently from other Q-values within the same state, Wang et al. (2016) describe a Q-value as the sum of the state-dependent value $V(s)$ and the advantage of each action in that state $A(s, a)$.

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

Formally, the state-dependent value function $V(s)$ is the expected Q-value received for an agent behaving according to a policy π as shown in Equation 7.

Algorithm 2 Double DQN Algorithm

```
1: Initialize replay memory $D$ to capacity $D_{capacity}$
2: Initialize action-value function $Q$ with random weights $\theta$
3: Initialize target action-value function $Q_{target}$ with random weights $\theta_{target}$
4: for episode $\in \{1, \dots, M\}$ do
5:     Observe initial state $s_1$
6:     Preprocess $s_1$ through $\phi$ to get $\phi_1$
7:     for $t \in \{1, \dots, T\}$ do
8:         Sample action $a_t \sim \pi$
9:         Execute action $a_t$ in the environment and observe reward $r_t$ and image $s_{t+1}$
10:        Preprocess $\phi_{t+1} = \phi(s_{t+1})$
11:        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$, replacing the oldest experience if $|D| \geq D_{capacity}$
12:        Sample random batch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $D$
13:        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma Q_{target}(\phi_{j+1}, argmax_{a \in A} Q(\phi_{j+1}, a)) & \text{for non-terminal } \phi_{j+1} \end{cases}$
14:        Calculate $loss_j = y_j - Q(\phi_j, a_j; \theta)$
15:        Combine losses with mean-squared function to obtain $loss$
16:        Use $loss$ to perform a gradient descent step on the Q-estimator
17:        Replace target parameters $\theta_{target} \leftarrow \theta$ every $\tau$ steps
```

$$V^{\pi}(s) = \mathbb{E}_{a \sim \pi(s)}[Q^{\pi}(s, a)] \tag{7}$$

Therefore, Wang et al. (2016) utilizes the two estimators $V(s)$ and $A(s, a)$ in their approximator for $Q(s, a)$ as shown in Figure 8.

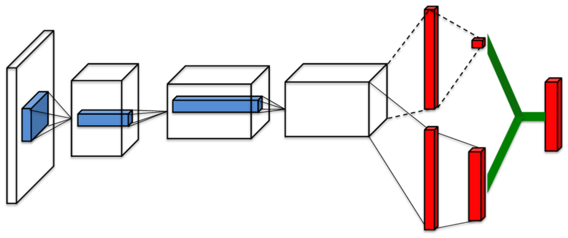

Figure 8: Dueling Architecture Diagram by Wang et al. (2016)

Intuitively, the advantage function describes the impact of each action on the expected return. The benefit of separating these estimators in the Q-network is that it allows us to generalize learning across actions. With every update to the Q values in this architecture, the value estimator V is updated. This contrasts with the previous approaches where only the value for one state-action pair is updated while the other state-action pairs within the state remain untouched.

This architecture also has the benefit of learning whether or not its actions have any effect on the environment in a given state. For example during a game of Pong, if the ball is moving towards the opponent, it does not actually matter how the agent moves the paddle at that moment in time. It only matters when the ball coming toward the agent's paddle as the paddle would need to be positioned properly to counter it.

To produce the Q-value, a special aggregating function is required to combine both the state-dependent value and advantage streams. It might be tempting to use the definition at the beginning of this section to obtain the Q-value, however, this makes the Q-value unidentifiable. Unidentifiable in this case means that given a Q-value, it is not possible to extract the state-dependent value or advantage from it. In practice, this leads to poor performance. To get around this, Wang et al. (2016) introduce the following combining layer shown in Equation 8 at the risk of introducing a small bias.

$$Q(s,a) = V(s,a) + (A(s,a) - mean(A(s))) \tag{8}$$

This layer forces the advantage function estimator to have zero advantage at the chosen action. In practice, this improves the stability of the algorithm.

Related Work

Interactive demonstrations borrow from aspects of imitation learning and other methods of incorporating demonstration data into a reinforcement learning algorithm. This section will go over the different related ideas and provide some commentary on potential issues with these approaches and how some parts are incorporated into Interactive Demonstrations.

Imitation Learning

Imitation learning is a class of algorithms that utilizes expert demonstration data in order to derive a policy that copies or imitates the expert's actions. A typical approach to imitation learning is to train a classifier to predict an expert's action at a given state. However, Ross, Gordan, and Bagnell (2011) note that the learner's predictions affect future observations of the learned policy. This violates the i.i.d. assumption as described earlier.

Ross, Gordan, and Bagnell (2011) state that given a probability of the classifier making a mistake p_m, the total number of mistakes in expectation over T-steps is $T^2 p_m$. Intuitively this is because as soon as the agent makes a mistake, it will encounter different observations than under the expert demonstration.

This then presents a state distribution mismatch between the dataset that the expert provides and the actual states encountered by the agent. Part of the reason for this is that the expert typically does not provide actions for every state possible in the environment. For example, suppose we were to collect a dataset on human driving. We then provide a suite of sensors to a good representative population. By this, we would think that a wide distribution of states would be collected. However, this is not always the case. Human drivers normally do not experience adverse driving conditions such as strong storms. The agent will then be unable to "imitate" a behavior it has not seen before or seen only a few examples of.

DAgger (Dataset Aggregation) is a meta-algorithm (shown in Algorithm 3) designed to wrap around an imitation learning algorithm in order to mitigate the state distribution mismatch. It does this by having the agent collect states within an environment and having the expert annotate which actions it would have performed in those states.

This method makes for an awkward implementation since a demonstrator would have to look at a random sequence of states and come up with what it thinks the best action is for each of them. In the ALE, most states consist of four frames of video input. At 30 frames per second, a state would be approximately as long as a blink. For a human demonstrator, it would prove challenging to annotate actions for a series of blink observations.

Model Pretraining

A hidden task associated with many environments (as shown in Figure 9) is discerning what information in a state is important. In the literature this is called feature learning. The idea behind model pretraining

Algorithm 3 DAgger

```
1: Initialize replay memory $D$ to capacity $D_{capacity}$
2: Initialize $\hat{\pi}$ with random weights $\theta$
3: for $i \in \{1, \dots, M\}$ do
4:     Observe initial state $s_1$
5:     Initialize state buffer $B \leftarrow \{s_1\}$
6:     for $t \in \{1, \dots, T\}$ do
7:         Sample action $a_t \sim \hat{\pi}$
8:         Execute action $a_t$ in the environment and observe image $s_{t+1}$
9:         Store $s_{t+1}$ in buffer $B$
10:    Initialize episode memory $D_i$
11:    for $s \in B$ do
12:        Sample expert action $a_s \sim \pi_{demo}$
13:        Store $(s, a_s)$ in $D_i$
14:    Aggregate datasets $D \leftarrow D \cup D_i$ and eject the oldest tuples while $|D| \geq D_{capacity}$
15:    Train classifier $\hat{\pi}$ on $D$
```

by Cruz Jr, Du, and Taylor (2018) (outlined in Algorithm 4) is to use demonstration data to learn these features. As hinted by the name, these are steps executed before the reinforcement learning algorithm begins. Therefore, the reinforcement learning algorithm can focus on policy learning without having to deal with feature learning, thus speeding up the learning process.

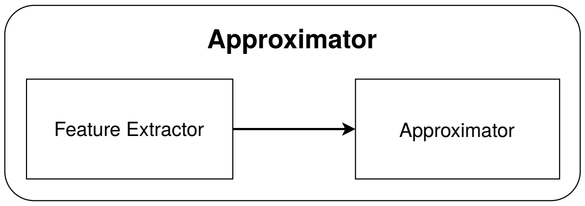

Figure 9: Hidden Component of Approximator

Algorithm 4 Model Pretraining

```
1: Gather demonstration data $D = \{(s_1, a_1), \dots, (s_k, a_k)\}$
2: Initialize approximator $f$ with weights $\theta$
3: Initialize $m \leftarrow 1$
4: for $i \in \{1, \dots, N\}$ do
5:     Sample batch from $D$
6:     Train classifier $f$ with batch while updating $m = max(m, f_{output})$
7: Divide parameters of $f$ by $m$, $\theta \leftarrow \frac{\theta}{m}$
8: Perform Reinforcement Learning Algorithm with approximator $f$
```

Model pretraining does not attempt to produce an accurate approximator, but instead tries to make the first few layers of the neural network an adequate feature extractor. It does this by training the approximator f to be a classifier similar to imitation learning. The idea is that features needed for an imitation learning algorithm are the same features needed for the reinforcement learning problem. The loss function used is Cross-Entropy (CE) (Equation 9) with the truth value y and the estimated value $\hat{y}$. During the pretraining process, the values in the output layer can grow significantly. Therefore at the end of this process, the weights are divided by the maximum value outputted by the approximator.

$$loss_{CE} = -y \log \hat{y} - (1-y) \log (1-\hat{y}) \tag{9}$$

Loss Manipulation

Initially training a Q-value approximator into a stochastic policy isn't the most ideal. Therefore, Hester et al. (2018) created the supervised Q-value margin loss to have a similar affect while maintaining a Q-value-like function. It works by making the Q-values of the expert states and actions, slightly higher than the rest in the state as shown by Equation 10.

$$J_E(Q) = max_{a \in A}[Q(s,a) + l(a_E, a)] - Q(s, a_E) \tag{10}$$

where $l(a_E, a)$ is a margin function that is 0 when $a = a_E$ and a small positive constant otherwise.

In the model pretraining algorithm, only the states and actions from the demonstrator are needed. However in Deep Q-Learning from Demonstrations (DQfD) the whole experience tuple (s_t, a_t, r_t, s_{t+1}) is needed from the demonstrator. In this approach there are a total of four losses to be considered:

1. 1 step double Q-learning loss ($loss_{TD}$ in Equation 3)
2. N step double Q-learning loss ($loss_{NTD}$)
3. Supervised Q-value margin loss ($loss_{margin}$ in Equation 10)
4. $L2$ regularization loss on weights and biases ($loss_{L2}$)

Going into the specifics of (2) and (4) are outside of the scope of this paper. However in summary, (2) uses the Bellman Equation to not only unfold the Q-value once, but N times. Meanwhile, (4) attempts to minimize the parameters of the approximator and lower the chances of overfitting. The word "double" in (1) and (2) means that the double DQN enhancement is used. All of the losses get applied if we are looking at the demonstration data. Meanwhile if the data is gathered by the agent, then all but (3) are applied.

$$loss_{demo} = \mathbb{E}[\lambda_{TD} loss_{TD} + \lambda_{NTD} loss_{NTD} + \lambda_{margin} loss_{margin} + \lambda_{L2} loss_{L2}]$$

$$loss_{agent} = \mathbb{E}[\lambda_{TD} loss_{TD} + \lambda_{NTD} loss_{NTD} + \lambda_{L2} loss_{L2}]$$

$$loss = loss_{demo} + loss_{agent}$$

where the λs are hyperparameters signifying the importance of each loss.

Demonstration data in the replay buffer is never overwritten. Instead, the agent only overwrites the data it collected itself during the learning process. Deep Q-learning from Demonstrations uses prioritized sampling as opposed to random sampling. Prioritized sampling uses TD loss (from Equation 3) as the weights when sampling from the replay buffer. Small positive constants ϵ_a and ϵ_d are added to the sampling weights in order to control the relative sampling of agent-gathered data versus demonstration data respectively.

Interactive Demonstrations

Interactive Demonstrations features a shared environment between the agent and the demonstrator. This interactivity allows the demonstrator to correct deviations from the task that an agent makes. In this way, the agent gets exposure to a wider variety of states than if the agent was learning from static demonstrations or on its own. Therefore, this results in a more robust approximator. In addition to the shared environment, the agent also trains on the side using a separate environment called a *hidden* environment. This is for

Algorithm 5 Deep Q-Learning from Demonstrations

```
1: Initialize replay memory $D$ to capacity $D_{capacity}$ with demonstration data
2: Initialize action-value function $Q$ with random weights $\theta$
3: Initialize target action-value function $Q_{target}$ with random weights $\theta_{target}$
4: for steps $t \in \{1, ..., k\}$ do
5:     Sample batch from $D$ with prioritization
6:     Calculate $loss_{demo}$
7:     Use $loss_{demo}$ to perform a gradient descent step on the $Q$-estimator
8:     if $t \bmod \tau = 0$ then
9:         Update target network, $\theta_{target} \leftarrow \theta$
10: for episode $\in \{1, ..., M\}$ do
11:     Observe initial state $s_1$
12:     Preprocess $s_1$ through $\phi$ to get $\phi_1$
13:     for steps $t \in \{1, ..., T\}$ do
14:         Sample action $a_t \sim \pi$
15:         Execute action $a_t$ in the environment and observe reward $r_t$ and image $s_{t+1}$
16:         Preprocess $\phi_{t+1} = \phi(s_{t+1})$
17:         if $|D| = D_{capacity}$ then
18:             Replace oldest self-generated experience in $D$ with $(\phi_t, a_t, r_t, \phi_{t+1})$
19:         else
20:             Add $(\phi_t, a_t, r_t, \phi_{t+1})$ to $D$
21:         Sample batch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $D$ with prioritization
22:         Calculate $loss = loss_{demo} + loss_{agent}$
23:         Use $loss$ to perform a gradient descent step on the $Q$-estimator
24:         Replace target parameters $\theta_{target} \leftarrow \theta$ every $\tau$ steps
```

several reasons. First, the agent can interact with an environment faster if it does not need to switch off with the demonstrator. This results in quicker feedback on the agents performance at the task. Next, the step where the approximator gets updated is time intensive. Updating the approximator when the agent is performing tasks in the shared environment will result in a jagged experience to a human demonstrator watching the agent.

From the introduction, common components of a reinforcement learning algorithm include: memory, approximator, loss function, and policy evaluation. We will quickly go over these different components and then describe how they all work together to form a cohesive algorithm.

Memory We modify the replay buffer to separate demonstrations versus agent-gathered experiences as in the DQLfD algorithm. However, since we are constantly gathering demonstrations in this approach, we set $D_{demomax}$ to be the maximum number of demonstrations to be stored in the buffer. Adding a new agent-gathered experience would only overwrite the oldest agent-gathered experience if the replay buffer is full. Meanwhile, adding a demonstrator experience when the replay buffer is full will try to overwrite the oldest agent-gathered experience unless $D_{demomax}$ is met. In the long run, this results in a consistent ratio between agent-gathered experiences and demonstrator experiences.

Approximator We wish to approximate the Q-value function with this approach.

Loss Function We will only use two different types of losses: double TD loss (Equation 3) and margin loss (Equation 10). The margin loss gets used in addition to the TD loss when calculating the loss of the demonstration experiences. Meanwhile, only the TD loss is applied to agent-gathered experiences.

$$loss_{demo} = \mathbb{E}[\lambda_{TD} loss_{TD} + \lambda_{margin} loss_{margin}]$$

$$loss_{agent} = \mathbb{E}[\lambda_{TD} loss_{TD}]$$

$$loss = loss_{demo} + loss_{agent}$$

Policy When operating in the hidden environment, we will use epsilon-greedy in order to encourage exploration. While interacting in the shared environment, the agent will use the greedy policy to give the demonstrator feedback on its progress. The policy used after training will be the greedy policy.

Training Process

The demonstrator will begin the training process by interacting with the environment for t_{play} seconds. Each experience gathered by the demonstrator is added to the replay buffer D. Then, the agent will start up a separate environment and interact with it M times through adding its experiences to D. Throughout each episode, the agent will every *skip* steps sample a batch from D and calculate a *loss* to improve the approximator with. After these interactions, the agent will take what it learned and evaluate its policy on the shared environment for t_{play} seconds taking over where the demonstrator left off. This process is then repeated until the demonstrator determines that the agent has learned enough through this process. The training process is further outlined in Algorithm 6.

Algorithm 6 Interactive Demonstrations

```
 1: Initialize replay memory $D$ to capacity $D_{capacity}$ with demonstration data
 2: Initialize action-value function $Q$ with random weights $\theta$
 3: Initialize target action-value function $Q_{target}$ with random weights $\theta_{target}$
 4: while training do
 5:     for $t \in \{1, ..., t_{play}\}$ do
 6:         if episode in shared environment finished then
 7:             Restart environment and set $s_t$ to the initial observation
 8:             Preprocess $s_t$ through $\phi$ to get $\phi_t$
 9:         Demonstrator provides $a_t$ given $\phi_t$
10:         Execute action $a_t$ in the environment and observe reward $r_t$ and image $s_{t+1}$
11:         Preprocess $\phi_{t+1} = \phi(s_{t+1})$
12:         if $|D| = D_{capacity}$ and $|D_{demo}| = D_{maxdemo}$ then
13:             Replace oldest demonstration experience in $D$ with $(\phi_t, a_t, r_t, \phi_{t+1})$
14:         else
15:             if $|D| = D_{capacity}$ then
16:                 Replace oldest agent-gathered experience in $D$ with $(\phi_t, a_t, r_t, \phi_{t+1})$
17:             else
18:                 Add $(\phi_t, a_t, r_t, \phi_{t+1})$ to $D$
19:     for episode $\in \{1, ..., M\}$ do
20:         Observe initial state $s_1$ in hidden environment
21:         Preprocess $s_1$ through $\phi$ to get $\phi_1$
22:         for $t \in \{1, ..., T\}$ do
23:             With probability $\epsilon$ select a random action $a_t$
24:             otherwise select $a_t = max_a Q^*(\phi(s_t), a; \theta)$
25:             Execute action $a_t$ in the environment and observe reward $r_t$ and image $s_{t+1}$
26:             Preprocess $\phi_{t+1} = \phi(s_{t+1})$
27:             if $|D| = D_{capacity}$ then
28:                 Replace oldest agent-gathered experience in $D$ with $(\phi_t, a_t, r_t, \phi_{t+1})$
29:             else
30:                 Add $(\phi_t, a_t, r_t, \phi_{t+1})$ to $D$
31:             if $t \mod skip = 0$ then
32:                 Sample random batch of transitions $(\phi_t, a_t, r_t, \phi_{t+1})$ from $D$
33:                 Calculate $loss_{demo}$ from the demonstrations in batch
34:                 Calculate $loss_{agent}$ from the agent-gathered experiences in batch
35:                 Combine $loss_{agent}$ and $loss_{demo}$ to obtain $loss$
36:                 Use $loss$ to perform a gradient descent step on the $Q$-estimator
37:     for $t \in \{1, ..., t_{play}\}$ do
38:         if episode in shared environment finished then
39:             Restart environment and set $s_t$ to the initial observation
40:             Preprocess $s_t$ through $\phi$ to get $\phi_t$
41:         Sample $a_t \sim \pi(s)$
42:         Execute action $a_t$ in the environment and observe reward $r_t$ and image $s_{t+1}$
43:         Preprocess $\phi_{t+1} = \phi(s_{t+1})$
44:         if $|D| = D_{capacity}$ then
45:             Replace oldest agent-gathered experience in $D$ with $(\phi_t, a_t, r_t, \phi_{t+1})$
46:         else
47:             Add $(\phi_t, a_t, r_t, \phi_{t+1})$ to $D$
```

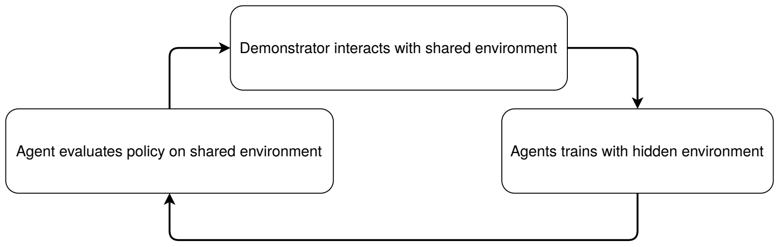
Demonstrator interacts with shared environment
Agents trains with hidden environment
Agent evaluates policy on shared environment

Figure 10: Training Flow Diagram

Experiment

Software

GymInteract is a custom library made using Python to sit on top of OpenAI Gym (Brockman et al. 2016). By itself, OpenAI gym is a programmatic API that allows agents to easily interact with the environment. In order for a human demonstrator to also interact seamlessly, additional software had to be written on top of this library. Using a software library called PyGame to output to the screen and capture input, Interactive Demonstrations' three stages were implemented into GymInteract. This library also features logging so that the experience tuples are regularly being logged to disk and a final video of the whole interaction is produced at the end.

RLTorch is a custom deep reinforcement learning library built upon the tensor library PyTorch. This library contains building blocks (such as memory, policy selection, etc.) that can be easily used to create a reinforcement learning algorithm. The algorithms listed in this paper have implementations in this library.

The custom code can be found on Github through the following URLs: https://github.com/brandon-rozek/gyminteract and https://github.com/brandon-rozek/rltorch.

Environment

The environment used to test Interactive Demonstrations is OpenAI Gym's implementation of Pong (Brockman et al. 2016). OpenAI Gym uses the ALE in the background to process inputs and states. It exposes a state as a Red-Green-Blue (RGB) pixel representation of the screen as shown in Figure 11. The screen is 210 pixels high and 160 pixels wide. The value of each RGB pixel is between 0 and 255.

The Atari 2600 (Figure 12) had a wide variety of supported controllers. The unit came packaged with a CX40 joystick (Figure 13), however it was recommended that Pong was played with a CX30-04 (Figure 14) paddle controller.

For each state, the agent has the following available actions: `NOOP`, `FIRE`, `RIGHT`, `LEFT`, `RIGHTFIRE`, `LEFTFIRE`. In Pong, firing is only relevant at the beginning as that starts the game. `RIGHTFIRE` is the equivalent to moving the joystick or paddle to the right and firing. `LEFTFIRE` is defined similarly.

A $+1$ is received when the agent (green paddle) scores the ball past the opponent's paddle, -1 is received when the opponent scores against the agent, and 0 is received otherwise. The game ends when either the agent or the opponent scores against the other player 21 times. The final score is produced by subtracting the opponents points from the agents as shown in Equation 11.

$$score = score_{agent} - score_{opponent} \tag{11}$$

Therefore the worst score the agent can obtain is -21 and the best score is 21.

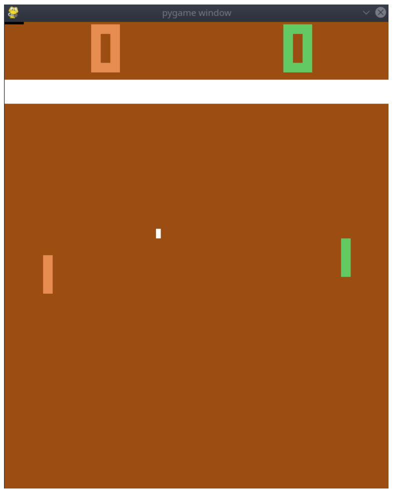

Figure 11: Pong from GymInteract

Figure 12: Atari 2600 Console (Amos 2019)

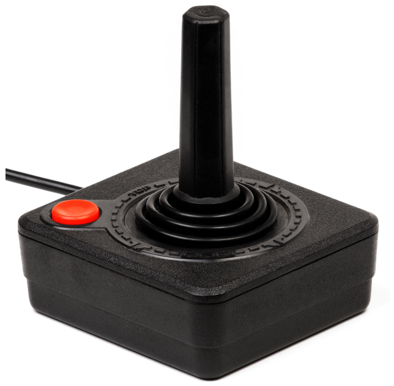

Figure 13: Atari CX40 Joystick (Amos 2010)

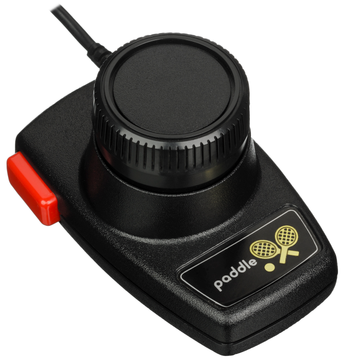

Figure 14: Atari CX30-04 Paddle Controller (Amos 2014)

By default, OpenAI's ALE implementations have a stochastic frame skip component to them. We disabled that by specifying `Pong-NoFrameskip-v4` as our environment.

Preprocessing

As mentioned in the introduction, we want the states presented to the agent to contain all the information the agent needs to make a decision and no irrelevant information. The reason we don't want extra information is that it the approximator has to learn that the extra information is unimportant. This would slow down learning. Therefore, we will preprocess the pixel screens received from the ALE into a 84x84x4 matrix. The preprocessing steps we will perform are taken from Dhariwal et al. (2017) They are listed in the order of which they are applied:

NoopResetEnv At the start of every game, we will perform no action for a certain number of frames. This will help us get a different sample of initial states from the environment.

MaxAndSkipEnv This step takes an action provided by the agent and execute it a fixed number of times in the environment. Pong is a game created for human consumption and humans don't make decisions in a frame by frame basis. Therefore, there is no need for our agent to make decisions like that as well. Having fewer decisions to make leads to having less to learn, which speeds up the process. This preprocessing step returns the sum of the rewards received during that time period and the maximum of the last two states received from the environment. This is one of the more impactful preprocessing steps given how much information is reduced by it. The number of frames that we do this for in Pong is `4`.

FireResetEnv Pong does not begin until the `FIRE` key is pressed. This could lead to wasted effort at the beginning of each game if the agent attempts to try other actions instead. Therefore, this wrapper automatically sends the `FIRE` key at the beginning of the game so that the agent does not have to explicitly do so.

ProcessFrame84 In the efforts to further reduce the amount of information that the agent needs to process, we note a few key ideas. For one, not all of the game screen is important to make decisions. For example, the score at the top of the screen isn't important in knowing where to move the paddle as well as the outer edges of the game screen. Therefore, we can crop those parts out and only focus only on the center portion of the screen. Color also isn't important in this game even though we noted the agent being the green paddle earlier. This is because the green paddle is always on the right side of the screen. Thus, knowing the color is irrelevant allowing us to remove the color information by averaging the three RGB values. Finally, the convolutional layers in a deep neural net approximator prefers square inputs, so we can downscale the remaining pixels to a 84 pixel wide by 84 pixel high screen. This process produces what is shown in Figure 15.

TorchWrap OpenAI gym returns states in a numpy array format. This wrapper transforms the numpy array into a PyTorch array which is the library that underlies `rltorch`.

FrameStack This step returns the last fixed number of states of an environment. Referring back to Figure 15, it is actually difficult to tell which direction the ball is heading. Therefore, a single frame isn't sufficient to give the agent enough information to act upon. If you provide the last two frames, then you know the direction in which the ball is heading but not the speed. Empirically it was determined that having a state consists of four frames leads to the best performance by Mnih et al. (2013).

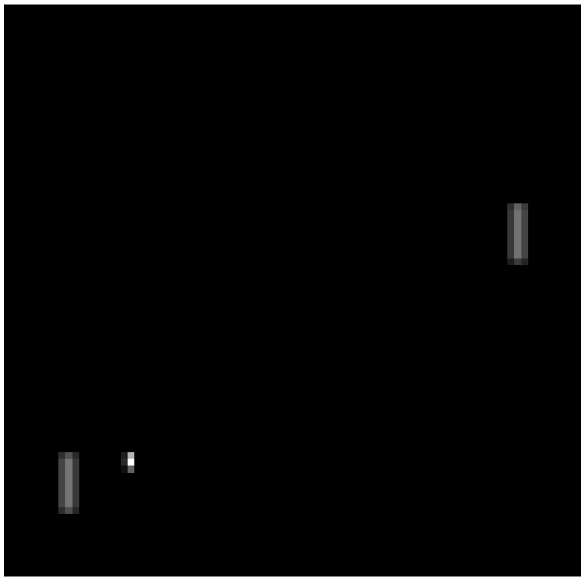

Figure 15: ProcessFrame84 applied to Pong

Results

To show the benefit of adding demonstrations throughout the learning process, we will compare Interactive Demonstrations to the Dueling Double Deep Q-Network (Dueling DDQN) approach. For simplicity, we'll refer to Dueling DDQN as DQN for the rest of this section. The hidden environment stage of Interactive Demonstrations is almost equivalent to the DQN algorithm if we ignore the change in loss function. As shown in Figure 16, Interactive Demonstrations starts beating the game, which is having a score above zero, faster than the DQN approach of having no demonstrations. Also, Interactive Demonstrations reach the maximum score of 21 faster than DQN as well. This shows the impact of adding demonstration data into a reinforcement learning algorithm.

Discussion

We tested this approach with a value based method, but there's nothing about the overall idea that's specific to this approach. In theory, a policy-based method can be used, however then the approximator, loss, and policy functions would have to be changed to accommodate that. The `GymInteract` software that accompanies this algorithm was designed for a human demonstrator, although this algorithm will work with a non-human demonstrator as well. This leads to use cases where an organization can spend a smaller amount of resources to create a simple agent through a logic-based approach. This agent can then be used as the demonstrator in Interactive Demonstrations. Though untested, better performance might be obtained if you set the importance of the demonstrators actions λ_{margin} to some sort of schedule that lowers the importance over time. The idea behind this approach is that there are a lot of behaviors to learn from the demonstrator at the beginning of training, however, the agent might start to become better than the demonstrator towards the end. This makes prioritizing the demonstrators actions in $loss_{margin}$ less than ideal.

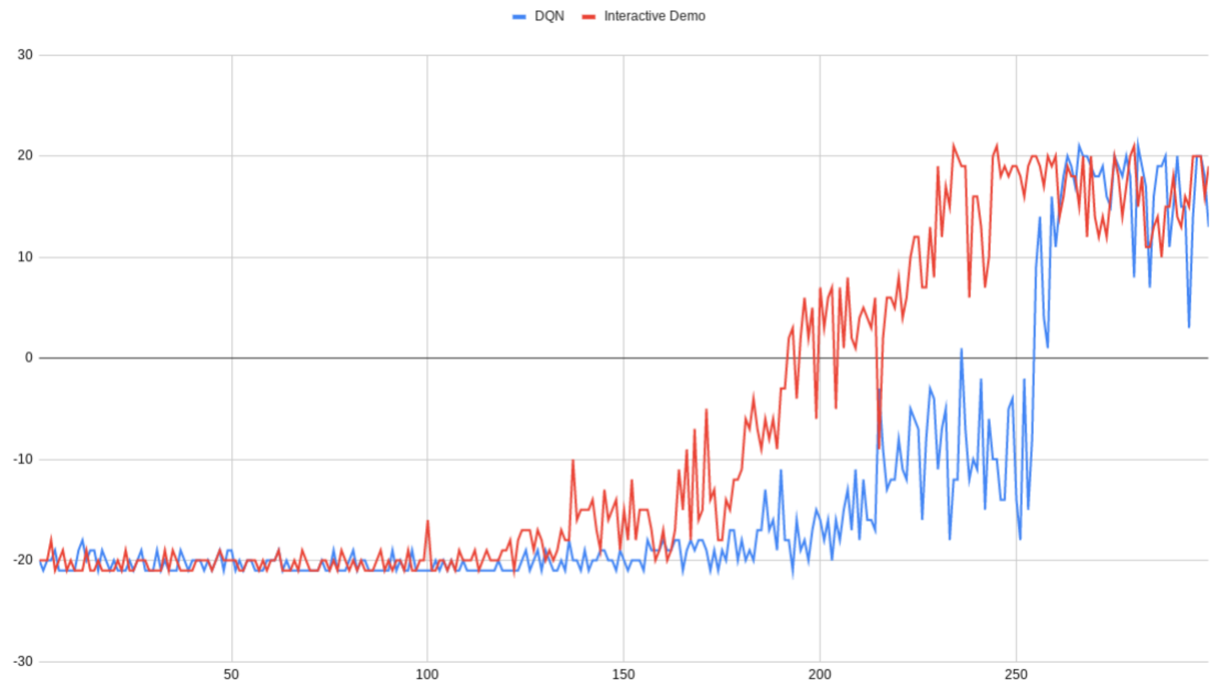

Figure 16: Comparison between Interactive Demonstrations and DQN

Acknowledgments

I would like to thank my research advisor Dr. Ron Zacharski for supporting me throughout this process and listening to my ideas. I would also like to thank my committee members Dr. Debra Hydorn and Dr. Stephen Davies for checking over my work and giving feedback and suggestions. Charlie Shields enabled my research to process video input thanks to his donation of a NVIDIA GTX 970 GPU, and Clare Arrington and Chris Richters helped polish my ideas and present this content in a more friendly way.

A. Definitions

Term	Definition
Action Selector	See Policy.
Advantage	Function that outputs the difference in return through one action compared to others in a given state.
Agent	Actor upon an environment.
Approximator	Structure that approximates a function.
Architecture	Topology or network structure of neural network.
Batch	Subset of experiences.
Bias	Accuracy or how wrong an approximator is on average.
Convolution Neural Networks	Neural network approximators that have convolutional layers which are used to process image input.
Deep RL	Reinforcement Learning with Deep Neural Networks

Term	Definition
Demonstrations	Tuples from a demonstrator such as (s_t, a_t) or (s_t, a_t, r_t, s_{t+1}).
Demonstrator	Another agent that provides example experiences for an environment.
Discount Factor	How myopic or short-sighted an agent is to future rewards.
Environment	The space in which actions are taken and states are observed.
Episode	Interaction with an environment until the end.
Experience	Record of a transition as (s_t, a_t, r_t, s_{t+1}).
Image	Observation released from the environment that is not a state.
Imitation Learning	Class of algorithms in which the agent attempts to copy the demonstrator.
Independent and identically distributed (i.i.d.)	The probability of sampling one experience is independent from sampling others.
Markovian	Property in which future states are only dependent upon the current one.
Memory	Buffer that stores transitions/experiences.
Model Free	Class of algorithms that do not learn a state transition function.
Off policy	Class of algorithms that use another policy in the loss function than the current one.
Optimization	A technique that tries to minimize or maximize the output of a function by changing its parameters.
Overfitting	Tailoring an approximator to a single observation at the cost of generalization.
Policy	Mapping from states to actions.
Preprocessing	The act of altering a state before passing it to a policy.
Q Function	Given a state and action, returns the expected discounted return.
Reinforcement Learning (RL)	The process of making sequential decisions to maximize reward.
Return	Sum of all rewards in an episode
Sample Efficient	Used to describe an algorithm in where an agent needs fewer interactions with the environment to learn a task.
Sparse rewards	Environments in where most transitions have zero reward.
State	A scenario from the environment that encodes information needed to make the best decision. Often used interchangeably with image.
State Dynamics	See State Transition Function
State Transition Function	Maps state-action pairs (s_t, a_t) to new state s_{t+1}.
Stochastic	Random in nature.
Tabula Rasa	Blank slate. Usually used to denote that the agent is learning from nothing.
Target Network	Stale copy of the current approximator. (See Double DQN Section)

Term	Definition
Transition	The act of moving from one state into another. Usually used interchangeably with experience.
Unidentifiable	A scenario in where not enough information is given to identify the piece desired.
Value-based method	Class of RL algorithms that tries to approximate the Q-value function
V Function	Given a state, returns the expected discounted return

B. Hyperparameters

Parameter	Value
Learning Rate, α	0.0001
Target Sync Rate, τ	0.001
Discount Rate, γ	0.99
Exploration Rate, ϵ	`ExponentialScheduler(start = 1, end = 0.02, iterations = 10^5)`
Replay Skip, *skip*	4
Batch Size, $batch_{size}$	$32 * (skip + 1)$
Number of Hidden Episodes, M	10
Seconds per Play, t_{play}	120
Memory Size, D_{size}	86400
Demonstration Loss Weight, λ_{demo}	0.01
Temporal Difference Loss Weight, λ_{TD}	1

Citations

Amos, Evan. 2010. Wikipedia. https://en.wikipedia.org/wiki/File:Atari-2600-Joystick.jpg.

———. 2014. Wikipedia. https://en.wikipedia.org/wiki/File:Atari-2600-Paddle-Controller-FR.jpg.

———. 2019. Wikipedia. https://en.wikipedia.org/wiki/File:Atari-2600-Console.jpg.

Bellemare, M. G., Y. Naddaf, J. Veness, and M. Bowling. 2013. "The Arcade Learning Environment: An Evaluation Platform for General Agents." *Journal of Artificial Intelligence Research* 47 (June): 253–79.

Bridle, J. S. 1990. "Training Stochastic Model Recognition Algorithms as Networks Can Lead to Maximum Mutual Information Estimates of Parameters." *Advances in Neural Information Processing Systems 2*, 211–17.

Brockman, Greg, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. "OpenAI Gym." *arXiv*.

Cruz Jr, Gabriel V. de la, Yunshu Du, and Matthew E. Taylor. 2018. "Pre-Training Neural Networks with Human Demonstrations for Deep Reinforcement Learning." *ALA*.

Dhariwal, Prafulla, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. "OpenAI Baselines." *GitHub Repository*. https://github.com/openai/baselines; GitHub.

Hasselt, Hado van, Arthur Guez, and David Silver. 2016. "Deep Reinforcement Learning with Double Q-Learning." *AAAI*.

Hester, Todd, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, et al. 2018. "Deep Q-Learning from Demonstrations." *AAAI*.

Levine, Sergey. 2017. "Supervised Learning of Behaviors." 2017. http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_2_behavior_cloning.pdf.

Lin, Long-Ji. 1992. "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning, and Teaching." *Kluwer Academic Publishers*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wlerstra, and Martin Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning." *NIPS Deep Learning Workshop*.

Polyak, B. T. 1990. "A New Method of Stochastic Appropximation Type." *Avtomatika I Telemekhanika*.

Ross, Stephane, Geoffrey J. Gordan, and J. Andrew Bagnell. 2011. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." *AISTATS*.

Ruppert, D. 1998. "Efficient Estimations from a Slowly Convergent Robbins-Monro Process." *Cornell University Operations Research and Industrial Engineering*.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. MIT Press.

Wang, Ziyu, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. 2016. "Dueling Network Architectures for Deep Reinforcement Learning." *International Conference on Machine Learning*.

Watkins, C. J. C. H. 1989. "Learning from Delayed Rewards." *PhD Thesis, Cambridge University*.