# Deep Reinforcement Learning and Experience Ranking on Atari Games

**John Salvador**
Auburn University
jms0256@auburn.edu

**Sri Ram Pavan Kumar Guttikonda**
Auburn University
szg0148@auburn.edu

## 1 Introduction and Motivation

Imitation learning (IL) in AI is when an agent learns from observing the actions of another. this is an approach to learning that we often see in nature. Many if not most important skills that we have were learned in-part through imitation learning. It was used when we learned to speak, drive, play music, and in many more. So because it is so important to natural learning it only makes sense to want to explore related techniques in AI. One such approach to imitation learning is through inverse reinforcement learning (IRL) which is a category of techniques that seek to infer the reward function from a set of demonstrations. The reason for this approach is that providing a set of demonstrations is much easier than having to design a reward function [1]. While this approach sounds promising, the issue with current techniques is that the performance of our agents are upper-bounded by the performance of the demonstrators. For our project, we seek to implement and observe the performance of the D-Rex algorithm [2] which is a technique of IRL that seeks to infer a reward function that allows our agent to learn to outperform the demonstrator that it learned from. Our implementation can be found at `https://github.com/jmsalvador2395/dl_project`

## 2 Related Work

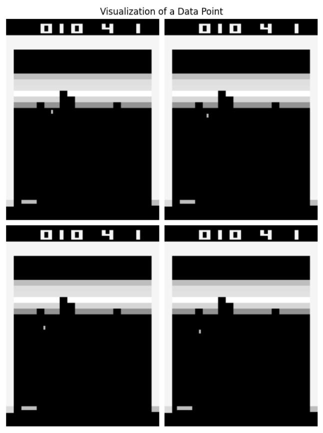
### 2.1 Deep Reinforcement Learning to Play Atari Games

Google DeepMind was able to showcase the power of deep learning when applied to reinforcement learning (RL) in their 2013 paper which showed state of the art performance on 6 Atari games [3]. Then only 2 years later in 2015, their published paper on DQN showed that they had achieved human-level performance on 29 out of 49 Atari games they had tested on [4]. A lot of progress has been made since then but the Atari games still remain a great benchmark for researchers to test their RL algorithms.

For reinforcement learning practitioners, OpenAI created the gym environment which provides a simple and convenient interface for us to implement our agents [5]. Because RL problems are almost always formulated as a Markov Decision Process (MDP), OpenAI's interface simplifies our code significantly and allows us to structure it to a standard format. While gym provides its own environments to practice RL algorithms on, its interface can be extended into other platforms. One such platform is the arcade learning environment (ALE) which lets us perform actions and capture data in supported Atari 2600 games [6].

### 2.2 Ranked Experience

Reward functions can be difficult to define and can be subjected to a lot of trial-and-error which, in the case of reinforcement learning, can take a long time to evaluate [7]. The intuition behind ranked experience is that, if we can show our agent examples of what we consider good and bad experience it would be able to infer, beyond the provided examples, what states we want to be in. Experience can be ranked in different ways like by total reward, or distance from the goal state. For our project



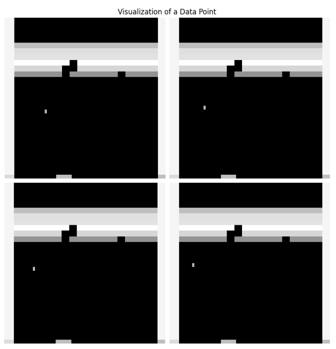


Figure 1: A visualization of one data point. Each frame is stacked on top of one another. Left is the original $4 \times 210 \times 160$ shape. Right is the cropped and scaled $4 \times 84 \times 84$ shape

and the paper we directly referenced [8], we somewhat take the approach of the former which we will explain in later sections.

## 3 Datasets and Methods

### 3.1 Data

Because our project is specifically in relation to our algorithm performance in Atari games, we use OpenAI's gym [5] and and the arcade learning environment [6] to aquire our data. We follow the practices of [4] where they use a preprocessing map $\phi(s)$ to reshape the data. Originally, we opted to use the raw $210 \times 160$ resolution grayscale images for our states but ran into issues with processing time and learning performance. Because of these issues we instead decided to follow the DeepMind's methods to crop and scale the image to $84 \times 84$ resolution. the game per frame and stack the last 4 frames on top of each other. A sample visualization of these four frames from our dataset can be seen in Figure 1. The idea behind this approach is that while this increases the dimensionality of the state space, the agent will be able to infer a direction of where objects are moving. In our project we implemented a class called data_collector which achieves this pre-processing and capturing of the images for our dataset. One other thing to note is that because the gym environment also implements frame skipping by default, our frames are technically separated by 4 times steps. That is, instead of having a stack $(f_t, f_{t+1}, f_{t+2}, f_{t+3})$, we have $s = (f_t, f_{t+4}, f_{t+8}, f_{t+12})$.

The data remains consistent but the labels change depending on which stage we are in the training process. For the behavior cloning section, our labels are the actions the demonstrator chose. For the reward function approximation section, the labels are either 0 or 1 depending on its index in the ranked demonstrations collection. Finally, for the deep q learning section, the labels are calculating the reward and Q value which we will discuss in the reinforcement learning algorithm section.

### 3.2 Problem Formulation

The goal of this project is to create a better-than-demonstrator performance given a set of trajectories generated by a better-than-random demonstrator. Our problem is modeled as a Markov Decision Process (MDP) which is a tuple $(S, A, P, r, \gamma)$ where $S$ is the set of states, $A$ is the set of actions, $P$ is the state transition probability function $P(s'|s) \in [0, 1]$ where $s', s \in S$. $r$ is the reward function $r(s, a) = \mathbb{E}[R_t | S_{t-1} = s, A_{t-1} = a]$ where $R$ is the reward at time $t$, $s \in S$, and $a \in A$. And $\gamma$ is the discount factor where $\gamma \in [0, 1]$.

As described in [8], we are focused on the problem of inverse reinforcement learning (IRL) where we don't have access to either the reward function $r$, nor do we have the rewards obtained by the demonstrator. Using the methods of [2], we take a set of what we assume to be better-than-random trajectories, run behavior cloning on them to obtain a policy $\pi_{bc}$ and then use that policy to generate a new set of ranked trajectories $\tau_t$ for $t = 1, ..., T$ where $\tau_i \prec \tau_j$ if $i < j$. Using these trajectories we approximate a reward function $\hat{r}$. Given $\hat{r}$ we then try to optimize a policy $\hat{\pi}$ that can perform better than the demonstrator.

### 3.3 Methods

As previously described, we use OpenAI's gym [5] and the arcade learning environment [6] to interface with our Atari games and collect our data. In our case we are only experimenting with the game Breakout. To collect the data we implemented a class called data_collector which takes the data, performs the function of $\phi(s)$ to shape our data as shown in Figure 1. We then play the game by ourselves to generate and collect the data that we need. After collecting the data we use PyTorch [9] to create a model and run behavioral cloning on the demonstrations to obtain our policy $\pi_{bc}$. Using $\pi_{bc}$ we use an epsilon-greedy method to inject gradually increasing noise, creating our ranked trajectories $\tau_t$. Using these ranked trajectories, we train a new network $\hat{r}(s)$ such that $\sum_{s \in \tau_i} \hat{r}(s) < \sum_{s \in \tau_j} \hat{r}(s)$. Finally, using our reward function $\hat{r}$, we train a policy $\hat{\pi}$ that ideally performs better than our demonstrations.

### 3.4 Gathering Demonstrator Data

The first step in our process is to gather demonstrator data. To do so, we created a data collector program that allowed us to play the game ourselves, collect the state-action pairs, and save them to our data directory after closing the game window. We chose to manually collect demonstrator data because at the time, we didn't have access to an agent that could automatically collect data for us. After implementing the deep q learning algorithm we had the ability to automatically generate data and even had the ability to change the skill level based on the model we used but that was a later milestone of the project that we couldn't wait for.

while playing the game, the data had to be cropped, resized, scaled, and then stacked before being added to our dataset. Thankfully, gym provided a convenient way to allow us to manually interface with the games and collect data using the `play()` function which takes a callback function as an arguement, allowing us to extract the data at each time step. Once again, the results of the image processing can be seen in figure 1

### 3.5 Behavior Cloning

Humans often learn how to perform tasks via imitation: they observe others perform a task, and then very quickly infer the appropriate actions to take based on their observations. We first take a set of unranked demonstrations and use behavioral cloning to learn a policy $\pi_{bc}$. Behavioral cloning treats each state action pair $(s, a) \in D$ as a training example and seeks a policy $\pi_{bc}$ that maps from states to actions. We model $\pi_{bc}$ using a neural network with parameters $\theta_{bc}$ and find these parameters using maximum-likelihood estimation such that $\theta_{bc} = \arg\max_\theta \Pi_{(s,a) \in D} \pi_{bc}(a|s)$. By virtue of the optimization procedure, $\pi_{bc}$ will usually only perform as well as the average performance of the demonstrator. at best it may perform slightly better than the demonstrator if the demonstrator makes mistakes approximately uniformly at random. In our project we have a demonstrator who performs a specific task and our AI looks at the demonstrator and try to behave similar to the demonstrator. We are using CNN, where we give the state pair and try to predict what action will be taken using a CNN. The input is a set of 4 images which represents the state of the system. We are taking a set of 4 images to see how the fire ball moves in the game. An action will be predicted with the help of CNN network.

Initially when we were running behavior cloning, we were getting poor results in both the classification task and in agent performance. Accuracy would only ever get 80% max and our agent would never do anything when evaluated. Bugs were found in the training code but the largest issue came from the balance of examples in the data set. After troubleshooting, we found out that our validation accuracy was directly related to the amount of `NOOP` actions we took when generating our data. This is the action for doing nothing and it made up a vast majority of the dataset. Because it made up

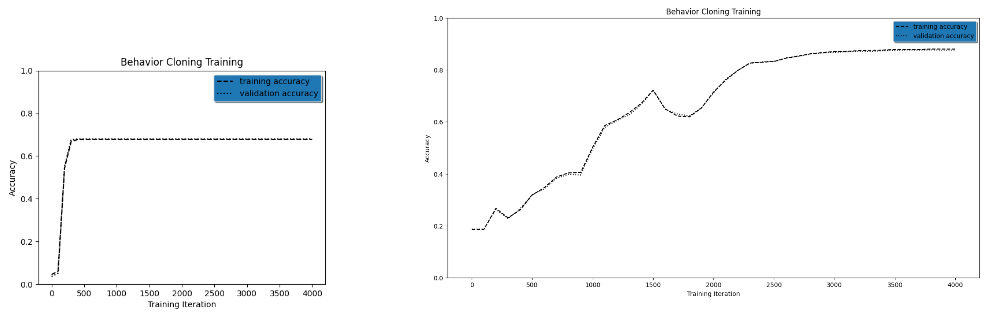


Figure 2: Behavior cloning accuracy. Left is before processing the data. Right is after fixing the distribution of labels in the dataset.

such a large portion of the data, the behavior cloning was biased towards the `NOOP` command and completely ignored the other commands. To fix this problem, we took the average of the amount of examples for all the other labels, and pruned our data so that the amount of `NOOP` examples left would be equal to that average. This left us with a lot less data, which required us to play more to get the same amount of samples. However, the balance of labels in the dataset fixed our issues and showed validation results that made much more sense while also resulting in a proper behavior cloned policy $\pi_{bc}$. The training results are shown in Figure 2

### 3.6 Disturbance-based Reward Extrapolation (D-REX)

D-REX is inspired from T-REX[8]. In T-REX, we have demonstrations ranked from worst to best $\tau_1, ...., \tau_m$. T-REX first performs reward inference by approximating the reward at state s using a neural network, $\widehat{R}_\theta(s)$, such that $\sum_{s\in\tau_i} \hat{R}_\theta(s) < \sum_{s\in\tau_j} \hat{R}_\theta(s)$ when $\tau_i < \tau_j$ . The reward function $\widehat{R}_\theta$ is learned via supervised learning, using a pairwise ranking loss based on the Luce-Shephard choice rule. After learning a reward function, T-REX can be combined with any RL algorithm to optimize a policy $\hat{\pi}$ with respect to $\widehat{R}_\theta(s)$. In T-REX we require a human to generate the ranked demonstrations, which is always not possible in the real world application scenarios. In D-REX, we avoid the requirement for humans to rank the demonstrations by considering the fact that $\pi_{BC}$ is significantly better than the performance of a completely random policy. We inject noise into $\pi_{BC}$ and compare the performance of $\pi_{BC}$ with a uniformly random policy generated by noise. given a noise set $\varepsilon = (\epsilon_1, \epsilon_2, ..., \epsilon_d)$ consisting of a sequence of noise levels such that $\epsilon_1 > \epsilon_2 > ..., > \epsilon_d$ then we can be assured with a high-probability that, $J(\pi_{BC}(.|\epsilon_1)) < J(\pi_{BC}(.|\epsilon_2)) < ... < J(\pi_{BC}(.|\epsilon_d))$. The noise levels $\epsilon \in \varepsilon$, are injected using an $\epsilon$ -greedy policy, where with probability 1-$\epsilon$ , the action is chosen according to $\pi_{BC}$, and with probability $\epsilon$ , the action is chosen uniformly at random within the action range. For every $\epsilon$ , we generate K policy rollouts or episodes and thus obtain K × d ranked demonstrations, where each trajectory is ranked from worst to best as noise decreases, that is initially we give a high noise which leads to low ranked demonstrations and then we decrease the noise to gather high ranked demonstrations. Thus, by generating rollouts from $\pi_{BC}(.|\epsilon)$ we get our ranked demonstrations as $D_{ranked} = \tau_i < \tau_j : \tau_i \sim \pi_{BC}(.|\epsilon_i), \tau_j \sim \pi_{BC}(.|\epsilon_j), \epsilon_i > \epsilon_j$. Given these ranked demonstrations, we then use T-REX to learn a reward function $\hat{R}$ from which we can optimize a policy $\hat{\pi}$ using any reinforcement learning algorithm.

There are certain challenges faced during the implementation of this D-REX algorithm. The main problem we encountered is the memory issues. Initially we considered storing 4 frames with size (4,210,160), when we are working on ranked trajectories this structure exceeded the 1MB limit of numpy array causing a halt to execution of program, so we changed the size to (4,84,84). The second problem is it takes a lot of time to get the ranked trajectories and train the reward network, it took us two days for this task. The authors ran 100 episodes for each noise value to get the ranked trajectories, whereas we were not able to run it greater than 30 episodes due to memory limitations. We feel that if we were able to gather more data for the reward network and have sufficient time to train it, the results would have been much better.

### 3.7 Deep Q Learning

While the papers we directly drew inspiration from utilized proximal policy optimization to train an agent using $\hat{r}$, we opted to use deep Q learning instead. Before being able to run the algorithm using $\hat{r}$, we first had to validate it's efficacy on smaller problems and reward function provided by gym. We tested the algorithm on the `Cartpole-v0` environment, using a basic 3 layer multi-layer perceptron and validated it to be effective at completing that task. However, adapting the algorithm for the `Breakout` environment was much more challenging for a few reasons. It was not immediately apparent to us how important all of the hyperparameters were when it came to this learning algorithm. Most notably, we hadn't thought about the importance of the frequency at whih to update the target network parameters to that of the policy network, and how often to update the policy network itself. In our tests for `Cartpole-v0`, we updated the target network every 10 training iterations which we assumed to be fine for `Breakout`. However, after some digging we found that in the provided code from DeepMind's 2015 paper, they updated the target network every 10000 training steps and updated the policy network every 4 [4]. Setting these parameters specifically and also changing the image resolution were the most important factors to getting this algorithm to work.

For the neural network architecture we went with how it was described in the original paper [3] [4]. The only difference in the architecture was the use of normalization layers and the initialization method. In our readings, we found recommendations to use the initialization method described in [10] because it allowed for more stable and consistent training. Because we originally wanted to work with the unprocessed $210 \times 160$ images, we had to alter the convolutional layers and also tried experimenting with more fully connected layers as well but due to terrible issues with with training stability, memory, and processing time, we had decided to go back to the $84 \times 84$ images described in the original paper to minimize the amount of differences between the original implementation and ours.

We also applied some suggested modifications to help with training time. First, because some blocks are worth more points than others, we decided to clip the reward to reflect the amount of blocks broken, rather than the score. It's arguable that this method may make it so that the agent takes longer to find the optimal policy but in our efforts to get this algorithm to work properly, we found this to work and decided to stick with it. The second modification was to condier a lost life as a terminal state. Doing this helps signal to the agent that a lost life isn't a desirable state, which helped training time and stability.

Improvements also had to be made to the algorithms memory usage and processing time. Initial implementations only allowed for 10000 replay memory examples which took up nearly 30 gigabytes of memory. There were a few improvements that let us cut back on that usage immensely. The first improvement was to remove unnecessary copy statements. This easily cut our memory usage in half. The second improvement was the change in resolution. The original $4 \times 210 \times 160$ samples took up almost five times as much memory as the $4 \times 84 \times 84$ samples which allowed for both better memory usage and faster processing time. The last improvement was on when to convert the data to torch tensors. Early implementations were converting data to numpy arrays and then to torch tensors. This has since been changed to go straight to torch tensors that are stored in cpu and then stacked and moved to gpu once a mini-batch has been sampled. These improvements allowed the algorithm to execute much faster than we had initially implemented it.

After validating the algorithm for the true reward function, substituting in $\hat{r}$ was simple. Instead of taking the true reward and adding that to our replay memories, we pass the resulting state $s^{'}$ to our reward network to obtain an output which we then feed into a sigmoid function before storing that value into our replay memories instead of the true reward function. Training results are shown in Figure 3.

## 4 Experiment and Results

We were only able to fit 2 training runs of deep Q learning on our reward network because the algorithm takes about 3 days in order to get results. Training history for experiment 2 is shown in Figure 3 and agent performance can be seen in figure 4. At the time of writing, the agent using $\hat{\pi}$ is growing but extremely slowly after being able to break four blocks. We were ultimately able to beat

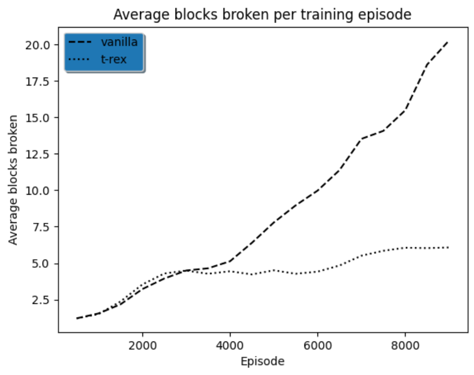


Figure 3: Average blocks broken for a vanilla deep Q learning agent and for the agent trained using $\hat{r}(s)$ over 9000 episodes of training

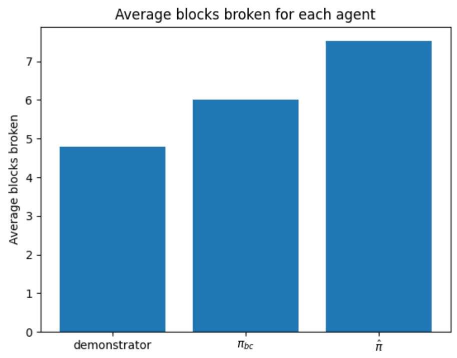


Figure 4: The average amount of blocks broken calculated for each agent

our demonstrator's average score of $4.8$ but the reasons for these results will be discussed in the next section.

## 5 Discussion and Conclusion

### 5.1 Discussion

As for why we were able to achieve a better-than-demonstrator, we believe that part of it was due to the special circumstances of how the reward function works and how the `Breakout` enviroment works as well. Because we feed the reward network output through a sigmoid function first, we always get a reward value at each time step that ranges from 0 to 1. Looking at it more closely, we noticed that the amount of reward we get per episode from $\hat{r}$ was always about 50 or 60 less than the length of the episode. We weren't able to verify which states the reward function evaluated lower but the main point that we are making is that the amount of reward goes up with the length of the episode. So while we believe our implementation is correct for the reward function approximation, it seems like it was only able to distinguish between preferential states up to the point where the agent performed like the demonstrator. After that we believe that improvements were only able to come from DQN training on terminal states because rewards were so consistent at not just the per-episode level but also the per-timestep level. Further experimentation into different environments and larger volumes of data would be necessary to be able to make better observations.

## 5.2 Conclusion

While were were able to achieve better-than-demonstrator performance with our new policy, it was not by much and nowhere near the results observed in [8] and [2]. There are many factors that could have contributed to this. The first reason we can think of is the data set size that we used for approximating the reward function. The second reason could be due to the reinforcement learning algorithm we chose. Deep Q learning is not state of the art and we also used the same hyperparameters that we used for deep Q learning with the true reward function. Different environments and implementation details need to be further explored but We were able to learn a lot about using the tools of reinforcement learning and deep learning along with the theory behind them as well.

## References

[1] Aaron Tucker, Adam Gleave, and Stuart Russell. *Inverse reinforcement learning for video games*. 2018. eprint: `arXiv:1810.10593`.

[2] Daniel S. Brown, Wonjoon Goo, and Scott Niekum. *Better-than-Demonstrator Imitation Learning via Automatically-Ranked Demonstrations*. 2019. arXiv: `1907.03976 [cs.LG]`.

[3] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: `1312.5602 [cs.LG]`.

[4] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 1476-4687. DOI: `10.1038/nature14236`. URL: `https://doi.org/10.1038/nature14236`.

[5] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[6] M. G. Bellemare et al. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *Journal of Artificial Intelligence Research* 47 (June 2013), pp. 253–279.

[7] A. Ng, Daishi Harada, and Stuart J. Russell. “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping”. In: *ICML*. 1999.

[8] Daniel S. Brown et al. *Extrapolating Beyond Suboptimal Demonstrations via Inverse Reinforcement Learning from Observations*. 2019. arXiv: `1904.06387 [cs.LG]`.

[9] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[10] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *CoRR* abs/1502.01852 (2015). arXiv: `1502.01852`. URL: `http://arxiv.org/abs/1502.01852`.