Integrating Deep Q-Learning with Learning from Demonstration to Solve Atari Games

Elizabeth Potapova
Binghamton University
epotapo1@binghamton.edu

Abstract

Deep Q-Learning has made leaps and bounds in efficacy and utility, yet it still requires a large amount of samples and time to reach a desired level of performance. Our project had the goal of patching up that weakness by integrating it with Learning from Demonstration. The goal was to inject the Deep Q-Learning model with human knowledge in order to jump-start the learning process. In this paper we introduce an agent that combines the two types of machine learning in order to play the Atari game Breakout. *Although we hypothesized that the training time would decrease for the Deep Q-Learning agent, our results show that the combined model trained for a longer time yet achieved a higher maximum score in the game compared to the plain Deep Q-Learning model.*

1. Introduction

Past advancements in creating intelligent agents have focused on creating a model that can achieve results on par with humans. IBM's *Deep Blue* was able to defeat the then-reigning champion of chess in a six-game match [4], and Google's *AlphaGo* conquered the reigning European champion at the game Go on a full board with no handicaps [18]. Yet, if one were to give those same agents a different, yet similar, task—say, playing checkers—then both models would fail at even the first step. Although these agents have mastered the discipline they were created for, they aren't applicable to diverse tasks.

Current advancements in machine learning focus on having an agent complete varying tasks using the same type of input. Some researchers have taken to training agents by having them play Atari games (Figure 1) [15]—old arcade machine games like *Pacman* or *Space Invaders*—which are perfect for training agents as they all share the same inputs but have drastically different game environments [1]. The team of Google's DeepMind created a Deep Q-Network (DQN) that was able to achieve performance on par with humans in playing different Atari games [14].

DQN combines Deep Learning [2] with Reinforcement Learning [19] (specifically Q-Learning [20]) in order to extract features from the Atari game environments and use them in order to achieve a maximum score. Given only pixels of the screen showing the Atari game, the model is able to see important features, making it the current state of the environment. Given the current state, possible actions, and possible future states, the model chooses an action that would maximize the reward, that being the score in the Atari game.

The process of choosing an action to maximize reward is not constrained to DQN, but Reinforcement Learning as a whole. Yet, to analyze how much a reward taking a specific action will yield, the agent needs prior knowledge of the outcome of taking such an action. For a Reinforcement Learning model in a new environment, there is an *exploration* phase that takes place in order to allow the model to evaluate rewards from taking actions in varying states. This phase of training requires a lot of data and, in turn, a lot of time in order for the Reinforcement Learning agent to perform well. For DQN, the agent requires hours of exploration and training in order to reach human-level performance.

One way to combat this would be to integrate a Learning from Demonstration (LfD) [17] model into the existing DQN model. Given a collection of labeled data, a LfD agent learns to label data of the same type correctly. In this case, given a frame of an Atari game being played, the LfD model outputs which action is best to take. Hypothetically, integrating an LfD model with a DQN model would decrease

Figure 1. Atari games, shown from left to right: *Breakout* (1976), *Space Invaders* (1978), *Pac-Man* (1980).

the data and time needed for the *exploration* phase, as the DQN can use the actions outputted by the LfD model instead of taking random actions. Thus, combining a DQN model with a LfD model can potentially reduce the training time for the DQN and produce human-level performance at a faster rate.

The paper describes such a combined model, yet the hypothetical outcomes were not achieved. Although the combined LfD-DQN model trained for a longer time than a plain DQN model, it was able to achieve a higher maximum score in the Atari game *Breakout*.

2. Related Works

Many machine learning algorithms evaluate their agent using the 1970's Atari games (Figure 1) due to their small screen output sizes and limited number of actions, which provide a simple yet effective challenge for artificial intelligence agents. These methods use OpenAI Gym [3], specifically the Atari environment; it utilizes the Atari Learning Environment [1] which is able to simulate hundreds of Atari 2600 video games and provide tools for evaluating and comparing machine learning agents.

Our main work derives from Google DeepMind's Deep Q-Network (DQN) algorithm, which is well-known for achieving scores in Atari games at the same or exceeding level than human players [14]. It utilizes Q-learning [20], a type of RL, to process pixels on screen and decide the best action to take to maximize reward.

Although DQN performed exceptionally well, it was hard to understand exactly which elements helped contribute to its level of success. A recent paper found that an agent that detected objects of similarly-colored pixels and how they moved through space and time aided it in achieving similar results to the DQN model [13]. The method was able to deduce if objects were essentially moving towards or away from the player.

Similarly, creating data structures for both models to see in the game environment is much more useful than raw pixel input [8]. Although a human can easily see a paddle and ball in the Atari game *Breakout* or the space ship in *Space Invaders*, a computer agent can make no such distinctions. Thus, implementing the method of determining where certain data structures lie, an agent performs significantly better than an agent without this implementation.

Other methods that have a machine learning agent play Atari games focus instead on improving the LfD model. These methods focus on guiding the training of the LfD model through trajectory preferences, meaning that human experts periodically let the agent know if it is learning the right thing and training in the correct direction, and the agent adjusts its training weights accordingly [11, 7]. Although this method does achieve higher results than a plain LfD model, we decided that we wanted to improve on the DQN model, whereas this method applies only to the LfD model.

A 2018 paper used LfD to solve hard exploration games that DQN had difficulty with, such as *Montezuma's Revenge*. These types of games require a long sequence of actions to be completed in order to receive the first reward, whereas RL excels at short-sequence actions. The authors created a method that starts right in front of the reward and "rolls back" the agent incrementally until reaching the beginning of the segment [16]. This model requires the agent to solve only one small step at a time, since when it reaches the state that the next step begins, it already knows which actions to take in order to receive the reward.

A similar approach of using LfD to accelerate the learning rate of an RL agent was proposed in 2018. It used a combination of temporal differences and supervised losses to pre-train the model based on demonstration data; it then updates its network based on interactions with the environment [10].

As our paper has the goal of improving upon the DQN algorithm by only integrating it with an LfD model, we are required to find an existing simple and effective DQN model as a baseline. One model was created in order to play the Atari game *Breakout*, which is the game our project focused on, and achieved a desirable score in a low amount of time [5].

3. Methods

In order to integrate the two types of machine learning, we first need to create the separate models for each and then combine them into one agent. First, the LfD agent needs to be created, which requires demonstrations, that being in-game frames of *Breakout* and the action taken at that frame. After collecting the required data, a model needs to be built, tuned, and trained to produce a saved LfD agent. Separately, the DQN model has to be set up for training and allowed to run without interruption to produce a saved DQN agent. Finally, we combine the two models by injecting the output of the LfD agent, that being an action to take, as a possible action to take for the DQN model chosen by random chance. After setting that up and allowing to train without interruption, we achieve a saved LfD-DQN agent.

3.1. Data Collection

Learning from Demonstration requires, as the name implies, human demonstrations for a computer agent to analyze and learn from. LfD can be applied to any type of labeled data or demonstrations; in the context of our project, these demonstrations are of us playing the Atari game *Breakout* and recording which action we took at which individual frame of the game.

Rather than having a script pause the game and wait for us to input a number correlating to an action for each frame, we found a simple keyboard agent script that assigns the action to the frame depending on what key is being pressed when that frame is visible (Algorithm 1) [9]. When no key is being pressed, the action defaults to No Action. Lastly, the images of each frame are stored in folders with names respective to those of all possible actions, with the name of the image being the time at which the frame was taken, using Python's time() function, to not have images replace one another.

Action	No Action	Spawn Ball	Move Right	Move Left	Total
Number of Images	11744	581	4508	4939	21772
Percentage	53.94%	2.67%	20.70%	22.68%	100%

Table 1. Breakdown of images of the game *Breakout* as per the action taken during that frame.

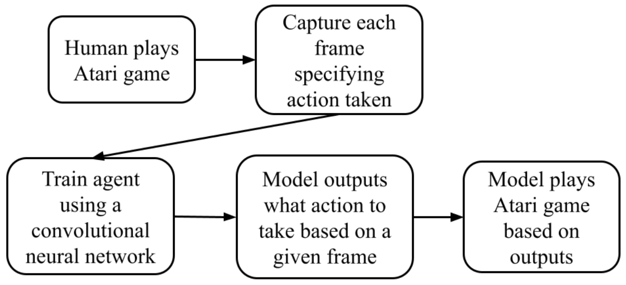
All collected images are organized as seen in Table 1. Although we were able to collect a decent amount of images for our LfD model to train from, the collected dataset is very skewed. About half of the collected images correlate to No Action, and the Spawn Ball action consists of less than 3% of the total data. Such an unbalanced dataset must be considered when being used to train the LfD agent.

Algorithm 1 Code chunk of script for data collection

```
Create Breakout environment
$action \leftarrow$ No Action
...
**if** key is pressed **then**
    **if** key pressed is 1 **then**
        $action \leftarrow$ Spawn Ball
    **else if** key pressed is 2 **then**
        $action \leftarrow$ Move Right
    **else if** key pressed is 3 **then**
        $action \leftarrow$ Move Left
    **end if**
**end if**
**if** key is released **then**
    $action \leftarrow$ No Action
**end if**
...
**while** game is running **do**
    $time \leftarrow$ current time
    Save PNG image of current screen with name of $time$
    in folder with name $action$
    ...
**end while**
...
```

3.2. LfD Model

Using the collected dataset for *Breakout*, we trained our LfD agent, the process of which is seen in Figure 2. Our agent is an image classification model that utilizes a convolutional neural network (CNN) [12] to analyze an image from the dataset and learn which class it belongs to. In the case of the game *Breakout*, given an image of a frame of the game the agent produces which action to take.



Human plays Atari game
Capture each frame specifying action taken
Train agent using a convolutional neural network
Model outputs what action to take based on a given frame
Model plays Atari game based on outputs

Figure 2. Process used to train a Learning from Demonstration model.

The code is written mainly using Keras [6], with a sprinkle of NumPy to calculate class weights. Before building the CNN model, the dataset needed to be preprocessed. All images were shifted into greyscale, as color data is not as relevant to the agent, and resized to a standard of 160 by 210 pixels. This was needed as some collected image were 320 by 210 pixels, most likely due to the different softwares that the data collection script was run on. Although the intended size of the Atari emulation is 320 by 210 pixels, it is best to use the smaller image size as to not clog up memory with increasing the image size. All pixel values within the image were normalized to a scale of 0-1 by dividing them all by 255, as greyscale values range from 0-255.

Next, the dataset is broken into two separate datasets, one for training and the other for validation. We used a standard split of 80% of all image for the training data, and 20% for the validation, or testing, data. A batch size of 32, seed of 123, class mode of categorical is used. There are 4 valid actions available in *Breakout*, those being 'No Action', 'Spawn Ball', 'Move Right', and 'Move Left'; these are the names of the 4 classes that the model categorizes images into.

The CNN is built using Keras' Sequential model, with layers of Conv2D, MaxPooling2D, BatchNormalization, Flatten, Dropout, and Dense (Algorithm 2). All of the convolution layers had a kernel size of 3 by 3 and used the ReLU activation function, and the number of filters increased from 16 to 32 to 64. Batch normalization and max pooling layers are inserted in between the convolution layers. Two of the convolution layers, one halfway through and one at the end of all of the convolution layers, had the L2 kernel regularizer set to 0.0001. The data is flattened, and 20% of learned data is dropped out. Finally, there are two fully-connected layers, the first one having 128 nodes with the ReLU activation function, and the final layers having 4

Algorithm 2 Convolutional neural network layers

```
Conv2D, 16 filters, 3x3 kernel size, ReLU activation
Conv2D, 16 filters, 3x3 kernel size, ReLU activation
BatchNormalization
Conv2D, 32 filters, 3x3 kernel size, ReLU activation
Conv2D, 32 filters, 3x3, ReLU, 0.0001 L2 regularizer
BatchNormalization
MaxPooling2D
Conv2D, 32 filters, 3x3 kernel size, ReLU activation
Conv2D, 32 filters, 3x3 kernel size, ReLU activation
BatchNormalization
Conv2D, 64 filters, 3x3 kernel size, ReLU activation
Conv2D, 64 filters, 3x3, ReLU, 0.0001 L2 regularizer
BatchNormalization
MaxPooling2D
Flatten
20% Dropout
Dense, 128 nodes, ReLU activation
Dense, 4 nodes, Softmax activation
```

nodes with the softmax activation function. The number of nodes in the last layer corresponds to the number of valid actions, and thus valid outputs, that the game has.

Both the kernel regularizer and dropout are means of preventing the model from overfitting to the training data and to learn more general patterns, thus decreasing the gap between training accuracy and validation accuracy. The model was compiled using the Adam optimizer and categorical crossentropy loss function.

Before running and training the model, the class weights for each action need to be calculated. This is because the dataset is highly skewed toward 'No Action' and away from 'Spawn Ball'. The weight for each class is calculated as such:

$$log\left(\frac{0.8 * \text{total number of images}}{\text{number of images in class}}\right)$$

Where any calculate weight below 1.0 is set to 1.0. Using the logarithmic scale smooths the weights down for very imbalanced classes. The number 0.8 is used for our dataset specifically, as it provides only one class that has a weight of 1.0; decreasing this value is useful for less imbalanced datasets. The weights for each class are then:

No Action	Spawn Ball	Move Right	Move Left
1.00	3.40	1.35	1.26

Table 2. Weights for each class used to train the LfD model.

As shown, the class with the most amount of images has the smallest weight, and vice versa. What this means is that during training, the model treats every instance of 'Spawn Ball' as 3.4 instances of 'No Action', and so on for each action.

The training data, validation data, class weights, and CNN model are combined and trained over 5 epochs. The LfD model achieved a training loss of 1.5456, training accuracy of 53.94%, validation loss of 1.1822, and a validation accuracy of 53.95%. The model trained within 4 minutes or 0.0667 hours (machine specifications are listed in 4.1). The final LfD model is then saved for testing.

3.3. DQN Model

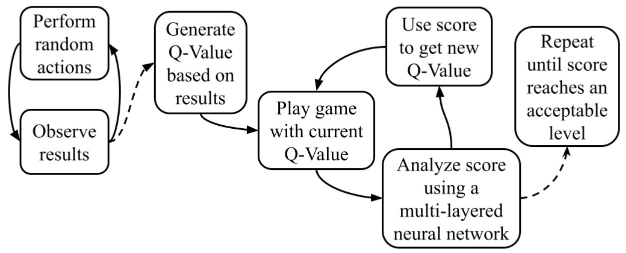

Perform random actions
Observe results
Generate Q-Value based on results
Play game with current Q-Value
Use score to get new Q-Value
Analyze score using a multi-layered neural network
Repeat until score reaches an acceptable level

Figure 3. Process used to train an agent by utilizing Deep Q-Learning.

Separately, we trained the chosen DQN algorithm [5] to play *Breakout*, the process of which can be seen in Figure 3. Deep Q-Learning uses a deep neural network to consider the current state, current possible actions, outcomes of actions taken in the past, and possible future states in order to determine a Q-value. This Q-Value is then used to determine which action will yield the most reward. More detailed information on how DQN works can be examined in Mnih et. al. [14].

Training starts off with the model taking a random action for 50,000 frames and observing the results. Then, the model switches to the *exploration* phase where it has 1,000,000 epsilon greedy frames; it starts off with epsilon, $\epsilon = 1$ and gradually decreases it until it hits a minimum of $\epsilon = 0.1$. If a randomly generated number from $0 - 1$ is within the range $(0, \epsilon)$, then the model takes a random action. To put it simply, the model starts off with a 100% chance to take a random action and gradually decreases that chance to 10% over the course of the epsilon greedy frames. If the randomly generated value falls outside of the range $(0, \epsilon)$, then the model a takes the action determined by the generated Q-value. It then observes the results and updates its weights accordingly. After reaching the maximum number of epsilon greedy frames, the DQN model switches to learning from actions taken solely on the generated Q-values.

The method used to determine if the agent is sufficiently trained is checking the average score over a portion of the latest games played. In the chosen DQN, the reward, or score, of the last 100 episodes, or games played, is saved into an array. The mean score over these last 100 episodes is computed, and this value is referred to as the running reward. In the original model we chose, the condition for completing training is checking to see if the running reward is above a 40. During training, although training speed was quick and memory not an issue, for some unknown reason the running reward would stagnate at around 10-11. With time constraints and unable to figure out why this was happening, we decided to change the condition for completion to be if running reward is over 10, which is a change in line 263 of the original model [5].

Some things that this version of DQN did differently from the DQN created by Google DeepMind [14] is that it decreased the size of memory that is available to the model during training and changed the optimizer for the model. Although we can only hypothesize, the limit on memory could have potentially caused the upper limit of 11 on the running reward during the training of our model. On the other hand, if this exact model was able to achieve a running reward of 40 within 24 hours for those researchers, perhaps the issue lies within software or hardware used on our machine.

We let the DQN model train without interruption, and it took a total of 10.467 hours to train (machine specifications are listed in 4.1). The trained DQN model is then saved for testing.

3.4. Combined LfD-DQN Model

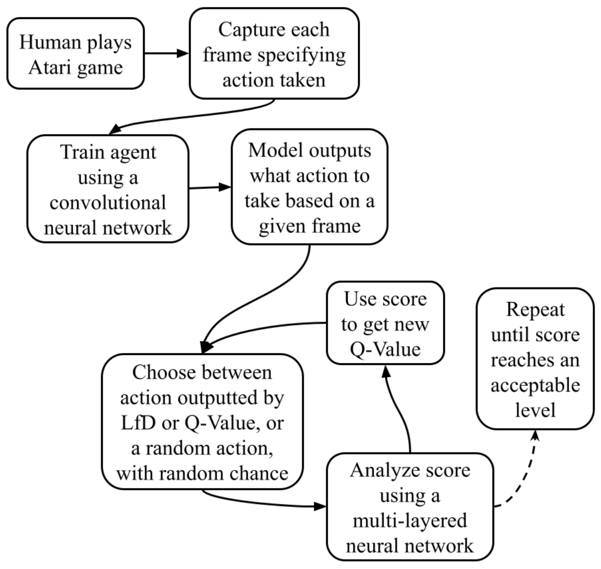

Human plays Atari game
Capture each frame specifying action taken
Train agent using a convolutional neural network
Model outputs what action to take based on a given frame
Use score to get new Q-Value
Repeat until score reaches an acceptable level
Choose between action outputted by LfD or Q-Value, or a random action, with random chance
Analyze score using a multi-layered neural network

Figure 4. Framework combining Learning from Demonstration and Deep Q-Learning.

After having both the trained LfD model and the trained DQN model complete, it is time to integrate the two into one combined model, the framework for which can be seen in Figure 4.

The method to do so would be to insert the action outputted by the LfD model as a possible action for the DQN to take during the epsilon greedy frames (Algorithm 3). That is, when the randomly generated value from 0-1 falls into the range $(0, \epsilon)$, there is a random chance for the DQN model to either take a random action or the action outputted by the LfD model. As our LfD agent received a lower than desired validation accuracy, we decided that this random chance should be split into 25%/75% to take the LfD action or a random action, respectively. Thus, when an initial randomly generated value from $0-1$ is within $(0, \epsilon)$, and a second randomly generated value from $0-1$ is within $(0, 0.25)$, then the DQN model executes the action outputted by the LfD model.

When such a chance is hit, the current frame of the given needs to be preprocessed before being passed into the LfD

Algorithm 3 Code chunk for integrating LfD into DQN

```
Create Breakout environment
Load trained LfD model
. . .
$\epsilon \leftarrow 1.0$
epsilon greedy frames $\leftarrow 1,000,000$
. . .
**while** training **do**
    **if** current frame $<$ epsilon greedy frames **then**
        $chance \leftarrow$ random number between $0 - 1$
        **if** $chance < \epsilon$ **then**
            $chance \leftarrow$ random number between $0 - 1$
            **if** $chance <= 0.25$ **then**
                Preprocess current frame
                Pass processed image into loaded LfD
                Take action generated by LfD model
            **else**
                Take random action
            **end if**
        **else**
            Take action generated by Q-value
            . . .
        **end if**
        . . .
        Decrement $\epsilon$
        . . .
    **end if**
    . . .
    **if** average score over last 100 episodes $>$ 10 **then**
        Training is complete
        . . .
    **end if**
**end while**
. . .
```

agent in the identical fashion to the images that the LfD agent was trained with. The screen of the game environment is saved, translated into greyscale, and rescaled to 160 by 210 pixels. Then the agent outputs the best action to take given that frame, and the DQN observes results as per usual.

We let the combined model train without interruption, and it took a total of 19.35 hours to train (machine specifications are listed in 4.1). The trained LfD-DQN model is then saved for testing.

4. Experiment

4.1. Testing

After creating and training all of the required models, the time comes to test them. We created scripts that load the saved models, played *Breakout* multiple times with different seeds, and kept track of which seed generated which score. Each agent played *Breakout* 5,000 times with a seed value generated randomly on a scale of 0 to 1,000,000,000.

One machine was used to train and test all of the models to ensure that hardware and software were not influential factors. This machine had a NVIDIA RTX 3080 as the GPU, Intel i5-10600K as the CPU, and 32 GBs of memory. Keras' GPU acceleration was utilized with NVIDIA's CUDA toolkit and cuDNN packages.

4.2. Results

	LfD	DQN	**LfD-DQN**
Training Time (hr)	0.067	10.47	19.35
Score/Accuracy	53.95%	41	43

Table 3. Training times of all models; validation accuracy for LfD and maximum scores achieved in *Breakout* for DQN and LfD-DQN.

The results of both training and testing can be seen above (Table 3), where the models were trained and tested without interruption. Since LfD achieved a score of 0 for any given seed, we decided that it would be a better representation of the model's efficacy. This result is most likely due to the little amount of images for the 'Spawn Ball' action, resulting in the model to learn it very poorly. We believe that the agent cannot spawn the ball on its own, but does a decent job determining actions to take given a frame already mid-game.

DQN achieved a best score of 41.0 at seed of 46936164. LfD-DQN achieved a best score of 43.0 at seed of 139126007.

5. Conclusion

We believe that the reason why the LfD-DQN model trained for a longer time than the plain DQN, against our prediction, is due to the fact that the LfD model achieved such a low validation accuracy. This potentially could have caused the longer training time as the DQN had to 'unlearn' the wrong actions that the LfD was outputting. Essentially, it comes down to the fact that it's worse to take the wrong action rather than a random action, as a random action could potentially be the correct action.

There are multiple ways to possibly improve the accuracy of the agent, which were touched upon in the Related Works section. One would be to introduce temporal data; as in, stacking frames to determine in which direction pixels are changing. This allows the agent to see what is changing with time and, for example, derive the trajectory of ball in *Breakout*. The DQN model already stacks 4 frames at a time before determining which action to take. If we can have the LfD model do the same process, perhaps the accuracy would drastically increase. One issue with this is that the data collected would have to be organized in a way for the convolutional neural network used to train the LfD agent to access random data in stacks of 4 images. As with our time constraints, having to re-capture the dataset for *Breakout* to facilitate this method would have cost us too much time.

Another possible method would be to break down the raw frame of the Atari environment into data structures, rather than looking at raw pixel information. For the game *Breakout*, this would consist of the ball, paddle, and blocks at the top. By having the agent determine these structures at the beginning of training, the agent would not have to determine these things on its own while training.

Without having to change the neural network for the LfD model, possible changes that would improve the LfD accuracy would be to collect a much more balanced dataset, or perhaps to not even consider 'No Action' as a class. That is, teach the agent to learn to always do something, rather than than do nothing in the environment. Another possibility is to limit how often the agent can use No Action as a choice.

The small increase in score compared to the large increase in training time could be a result of the weak LfD agent. Meaning, the integrating the LfD model made little to no impact compared to choosing a random action. To truly test the results of integrating the two machine learning methods, we believe the first step would be to increase the accuracy of the LfD agent. Further, experimenting with the percent chance of which action to take in order to see which would produce a better outcome; rather than the 25%/75% split we implemented for taking the LfD action/random action, respectively, future works can experiment with a 50%/50% split or a 80%/20% split.

References

[1] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation

platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013. 1, 2

[2] Yoshua Bengio. *Learning deep architectures for AI*. Now Publishers Inc, 2009. 1

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. 2

[4] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002. 1

[5] Jacob Chapman and Mathias Lechner. Deep q-learning for atari breakout. `https://github.com/keras-team/keras-io/blob/master/examples/rl/deep_q_network_breakout.py`, 2020. 2, 4, 5

[6] François Chollet et al. Keras. `https://keras.io`, 2015. 3

[7] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2017. 2

[8] Gabriel V. de la Cruz, Yunshu Du, and Matthew E. Taylor. Pre-training with non-expert human demonstration for deep reinforcement learning. *The Knowledge Engineering Review*, 34, 2019. 2

[9] Kevin Frans, Christopher Hesse, and Christopher Berner. `https://github.com/openai/mlsh/blob/master/gym/examples/agents/keyboard_agent.py`. 3

[10] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, et al. Deep q-learning from demonstrations. In *Thirty-second AAAI conference on artificial intelligence*, 2018. 2

[11] Borja Ibarz, Jan Leike, Tobias Pohlen, Geoffrey Irving, Shane Legg, and Dario Amodei. Reward learning from human preferences and demonstrations in atari, 2018. 2

[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 3

[13] Yitao Liang, Marlos C Machado, Erik Talvitie, and Michael Bowling. State of the art control of atari games using shallow reinforcement learning. *arXiv preprint arXiv:1512.01563*, 2015. 2

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015. 1, 2, 4, 5

[15] Yavar Naddaf. Game-independent ai agents for playing atari 2600 console games. *University of Alberta Libraries' Theses and Dissertations*, 2010. 1

[16] Tim Salimans and Richard Chen. Learning montezuma's revenge from a single demonstration, 2018. 2

[17] Stefan Schaal et al. Learning from demonstration. *Advances in neural information processing systems*, pages 1040–1046, 1997. 1

[18] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. 1

[19] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 1

[20] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992. 1, 2