Exploring Reinforcement Deep Learning with Atari

Russell Bisker

GMU CS 747 Spring 2025

Abstract

This project explores the application of Deep Q-Networks (DQN) to the Atari 2600 game Breakout, with a focus on implementing and extending the original DQN and Double DQN (DDQN) architectures. Using the Gymnasium framework and Arcade Learning Environment (ALE), I trained convolutional neural networks to predict action-value estimates from visual input alone. Multiple enhancements were explored, including a branched CNN with frame difference inputs and Prioritized Experience Replay (PER) with importance sampling. Although some of these additions showed limited improvement, the standard DQN and DDQN agents achieved performance exceeding baseline expectations. This project also surfaces practical challenges in reinforcement learning experimentation, such as training speed bottlenecks and infrastructure limitations.

1. Introduction

Reinforcement learning (RL) has emerged as a powerful framework for training agents to make sequential decisions through interaction with an environment. The combination of RL with deep learning—commonly referred to as deep reinforcement learning (DRL)—has enabled agents to achieve better than human levels of performance in complex tasks using raw visual input data. In particular, retro video games have proven to be a fertile domain to explore the possibilities of DRL Among the foundational breakthroughs in this area was the introduction of Deep Q-Networks (DQN) by Mnih et al. [1], which demonstrated that convolutional neural networks could learn effective policies from pixel data in Atari games. This was later refined by Hasselt et al. [2] through the use of Double DQN (DDQN), which reduced overestimation bias in value estimates.

In this project, I aimed to gain a hands-on understanding of DRL by building and experimenting with DQN-based agents to play the Atari 2600 game *Breakout* (see figure 1). Specifically, I focused on implementing and then attempting to improve upon versions of the agents and

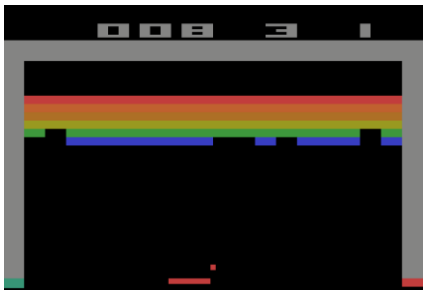

Figure 1: Screenshot from *Breakout* for the Atari 2600. The goal is to move the paddle at the bottom of the screen to bounce a ball at the colored "bricks" above. When a brick is hit by the ball, it disappears and score increases.

models discussed in the above papers. Everything was built making use of Gymnasium, a Python reinforcement learning framework that provides a standardized API to access the Arcade Learning Environment (ALE) [3], itself a low-level emulator for Atari 2600 games written in C++.

Beyond replicating the standard DQN and DDQN architectures, I explored various architectural and training modifications, including alternative loss functions, input representations, and experience replay strategies. While some extensions showed promise and others fell short, I was successful in creating game playing agents with a moderate – though significantly above "naïve" – degree of skill. They are likely not as proficient as the agents produced by the original researchers, primarily due to compute and training time constraints, though a 1:1 comparison is difficult due to variations in training and testing methodologies. I was nonetheless able to meaningfully exceed the performance benchmarks set in the homework assignment I used as a starting point.

2. Theoretical Background

For the standard DQN and DDQN I relied on the theoretical underpinnings introduced in the original papers, but for context, a brief summarization is provided in this section. *Breakout*, like any Atari game, consists of a series of frames where each image displayed on the screen, $x_{t,}$ is accompanied by an action a_t taken by the player. This combination could be considered a state in an of itself, but it lacks some important context, such as, in

the case of *Breakout*, the velocity and direction the ball is traveling. Thus, I instead define a state as a series (of history length, h) of frames and actions, such that $s_t = x_1, a_1, x_2, a_2, \dots, a_{h-1}, x_h$ A sequence of these states that terminates at a finite point makes up a Markov decision process (MDP), to which we can apply the well-known *Bellman equation*. This defines the value of taking a particular action at given state as $Q^*(s,a) = E_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') \,\middle|\, s,a \right]$, i.e. the Q-value of an action is the reward from that action plus the expected discounted value of the value maximizing action taken at the next state, and so on, until a terminal state is reached. Using traditional dynamic programming reinforcement learning algorithms (typically Value Iteration or Policy Iteration) this can theoretically be solved analytically. However, for even moderately complex games with large numbers of states and/or actions, this is practically and/or computationally intractable. This provides the motivation to instead use deep learning techniques to attempt to directly "learn" the Q-function by inputting (state, action) pairs into a neural network that outputs the Q-values for the entire set of actions that can be taken at that state.

2.1. Experience Replay

An obvious hurdle to implementing this as a training task is the lack of a readily identifiable "ground-truth" to compare learned Q-values against. If the correct Q-values for a state in a game were known, the game would already have a solved perfect strategy. Hence, deep reinforcement learning is inherently an unsupervised learning task. The solution is to use the neural network itself to generate the target values to train against. Prior to the Mnih paper, my understanding is that the typical approach was to use a network (the *target* network) with weight parameters updated in the prior iteration (θ_{t-1}) to compute the target, y_i. The time difference error (TD-error) can then be calculated as $\text{TD}_i = \left(y_i - Q(s,a;\theta_i)\right)$, where $y_i = \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \,\middle|\, s,a \right]$. One of key insights of the Mnih paper was to instead build an Experience Replay , a buffer of states collected during training as the agent was playing using an ε-greedy strategy (defined as taking a random action ε percent of the time and using the target network to find and take the max Q-value action (1- ε) percent of the time.

2.2. DQN vs Double DQN

The standard DQN process relies on sampling a random batch of states s_t and "next-states", s_{t+1}, from the replay buffer, passing the states to the current iteration of the network (the *policy* network) and the "next states" to the target network, then calculating a loss and performing stochastic gradient descent to update the policy network. The target network parameters are then updated to match the policy network periodically (the exact frequency is controlled via a hyperparameter typically in the range of once every 1K – 10K training steps)

While this works quite well, it does have the downside of creating positive feedback loops that lead to overestimated Q-values, due to both the action selection and value estimation being done using the same learned Q-function. The algorithm selects the maximum Q-value from noisy estimates, tending to select overestimated values which creates a bias that is then propagated into the learning target as well, causing the network to learn even higher Q-values.

To address this issue, the Hasselt et al. paper introduced the concept of a Double DQN. It still makes use of both a policy network and target network, but the key difference is that while actions are selected based on the Q-values calculated using the policy network, the training target is calculated using Q-values from the target network for that policy network selected action. The paper showed that this is successful in reducing overestimation bias (the mathematics of which are beyond the scope of this writeup), and it generally led to more successful agents (i.e. produced higher scores) across a variety of Atari games. Notably though, *Breakout* was one of a minority of games where the Hasselt paper actually reported higher results for the standard DQN – something I saw as well in my empirical results (to be discussed further later).

3. Implementation

As a starting point, I began with the code base skeleton that was included in the University of Illinois homework assignment [4] that I was referred to by Prof. Kosecka. My work for this project not only completed the tasks presented in the assignment by filling the necessary missing code, I made extensive revisions and additions to the point that I would estimate 60-70% of the final codebase was written or at least modified by me.

3.1. Game Environment

The original code utilized the OpenAI Gym library which is no longer maintained, so I migrated everything to Gymnasium [5], an up to date fork maintained by the Farama foundation. For the game environment, I used the ALE/Breakout-v5 environment to generate 210x160 pixel grayscale observations the game screen that I down-sample to 84x84 floating point arrays normalized to [0,1] for use throughout the code. Of note is that in their original form Atari games are entirely deterministic, meaning an agent could theoretically learn to memorize an optimal sequence of actions rather than actually learning and reacting to observations from the environment. To correct for this, ALE provides the option to set a repeat-action-probability parameter, which adds stochasticity to

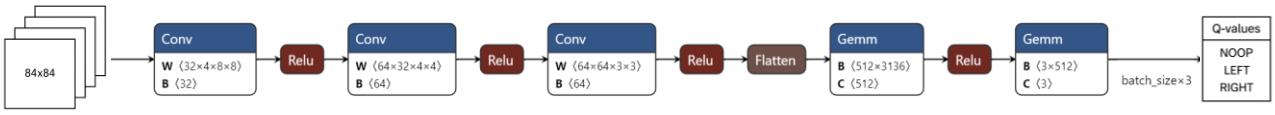

Figure 2: Basic CNN architecture

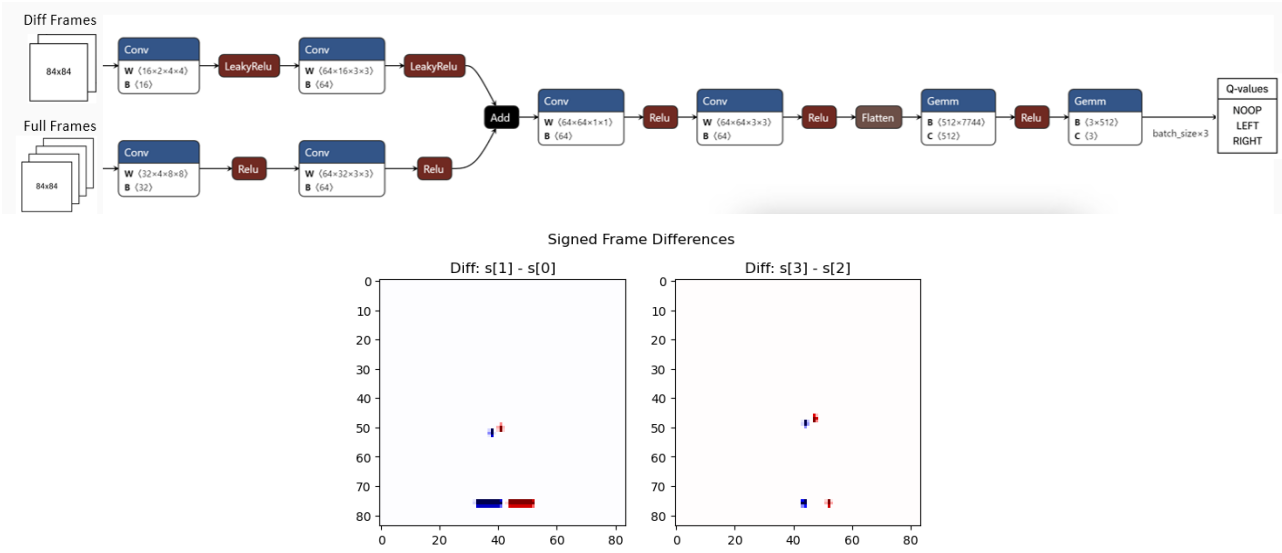

Figure 3: Top - Branched CNN architecture; Bottom – Example of two diff frames input to the top branch

gameplay by randomly repeating actions at the specified frequency. I experimented with both purely deterministic environments and low levels of repeat action probabilities (<25%). The most noticeable impact of this is changing the direction the ball launches on each new "life", which in turn can have a dramatic impact on the trained agent's strategy generalization.

Similar, to the DQN paper, to improve efficiency, I also used a frame-skip of 4, mean that every "frame" (as discussed elsewhere throughout this paper) is actually the result of the emulator running 4 frames with the associated action repeated in each frame.

I also configured the emulator to limit the action set to FIRE (launching the ball), LEFT, RIGHT (moving the paddle), and NOOP (doing nothing). However, I did not include FIRE in the set of trainable actions. In a normal round of 5 lives, the ball only needs to be FIRE-ed 5 times, so these transitions would make up an exceedingly small fraction of the replay buffer, making training difficult. Instead, I used metadata output from the emulator to detect when a life was lost, and then have the training/testing loop first perform a few randomized paddle movements (to add variability), before sending a FIRE action to start the next "life". I will note it was not entirely clear how this was being handled in the original assignment code, but I believe my implementation is much cleaner.

Accordingly, the Q-function I am training represents the current reward plus the discounted sum of all future Q-values resulting from optimal actions, from the current state until a "lost life" occurs (i.e. when calculating target Q-values, if the next state is a "lost life", the Q-value of the next state is 0). An episode is defined as 5 lives that each proceed until a "lost life".

3.2. Network Architecture

The assignment provided code for a basic CNN, which takes as an input the 84x84 grayscale frames for 4 consecutive states (due to the frame-skip these are actually the 4th, 8th, 12th, and 16th frames in a sequence from the emulator), and passes them through 3 convolutional layers and 2 fully connected layers to output Q-values for each of the 3 trainable actions (see fig 2).

I also built a second CNN (see fig 3), with a branched input structure. In addition to the 4 "full" frames, it accepts 2 "diff" frames that are calculated as $s_1 - s_0$ and $s_4 - s_3$. The aim here was to provide the network with images that isolated the movement of the ball and paddle, which in theory are the most important parts required to learn what actions to take. I experimented with a variety of activations functions for the diff branch but settled on Leaky ReLU (as opposed to standard ReLU used throughout the rest of the network), given that the diff frames consist of both positive and negative floats to distinguish movement direction. That said, despite multiple attempts at tweaking, the branched infrastructure

was infrastructure was not particularly successful (see Results section)

3.3. Loss Functions

In the original DQN paper, the authors used MSE (of the policy-net Q-values vs the target Q-values). However, based on the recommendation in the Univ. Illinois assignment and my own cursory research, I chose to implement a Huber loss function instead. I believe this has become the standard choice in deep RL research because it is more robust to outliers in TD-error, which stabilizes learning and reduces the possibility of Q-value explosion.

$$L_\delta(x) = \begin{cases} ½ \cdot x^2, & if\,|x| \le \delta \\ \delta \cdot (|x| - ½ \cdot \delta), & otherwise \end{cases}$$

Equation 1: Huber Loss

While the Huber loss (with $\delta = 1$) made sense when using a standard replay buffer, for comparison purposes, I also implemented a Prioritized Experience Replay (PER) combined with importance sampling (IS). In this case, a weighted version of MSE was used, because MSE interacts cleanly with importance sampling weights, preserving an unbiased estimate of the expected loss.

For PER, minibatch loss is calculated as:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} w_i \cdot \mathrm{TD}_i^2$$

Where:

- N is the batch size
- $w_i = \left(\frac{1}{N*P(i)}\right)^\beta$ is the IS weight
- $P(i) = \frac{p_i^\alpha}{\Sigma_k p_k^\alpha}$ is the probability of sampling transition i
- $p_i = |\mathrm{TD}_i|$ is the priority of transition i

Here, α determines how much prioritization is used, and β controls the strength of IS correction.

3.4. Other Improvements Explored

Over the course of working on this project, I implemented and experimented with a significant number of ideas, the majority of which unfortunately ended up bearing little fruit. While disappointing, these efforts still provided a valuable learning experience to better understand DQNs, PyTorch, and the ALE/Gymansium frameworks. Below is a non-exhaustive list of some of the things I tried.

- To try to improve training speed, I built a training loop that initialized parallelized game environments. Although I was able to get it working, it didn't meaningfully increase training speed because each environment still relied on the same neural networks which proved to be a bottleneck
- Rather than the default linear epsilon annealing, I tried other decay functions such as exponential and inverse time decay. I also tried applying one time epsilon "bumps" when score performance didn't show any improvement over several hundred episodes (to try force more exploring and try to escape strategies that worked well at first but not later in rounds).
- Instead of training every frame, I dynamically reduced the training frequency as episodes progressed.
- Reducing max replay buffer size (which is implemented as deque for the standard buffer and a circular list for the PER buffer), in an effort to ensure that earlier, short rounds where lives were lost quickly were replaced with more diverse gameplay of states where more blocks had been removed.
- Rather than simple diff frames calculated by subtraction, I tried using the cv2 library to calculate optical flow frames, but this ended up producing nearly blank images, due too little change in motion and texture occurring between frames (apparently the optical flow algorithm relies on detecting dense texture-based motion)

4. Challenges

By far the biggest challenge of this project was the immense amount of time completing required to complete training runs, combined with often being unable to get a sense of whether a change was beneficial or not and/or catch bugs until hours into a training run, and then being forced to restart. When combined with multiple network architectures and a multitude of hyperparameters – not only those typical for deep learning, but those specific to RL such a memory size, exploration/exploitation tradeoff, target network update frequency, and PER α and β – rigorous optimization testing became impossible given my time and resource constraints. These challenges were at least partially exacerbated by my relatively low level of experience with PyTorch, and to a lesser degree, Python in general (especially moderately large code bases). A lot of time could have been saved if, from the beginning, I had implemented better unit testing, debugging harnesses, logging, checkpointing, etc. to improve my end-to-end workflow. Instead, this all evolved as I went, but I certainly learned a lot about how to approach a project like this more efficiently from the start in the future.

Implementing the basic DQN and Double DQN with minimal changes to the stack that was originally provided in the homework assignment resulted in training runs of 750K steps taking 6-8 hours, which was long but manageable.

However, I faced significantly more issues when making meaningful architecture changes like the branched network and the PER buffer. For example, for a DQN with PER, training 550K steps took over 30 hours. I spent considerable effort trying to increase efficiency. For the PER code (where CPU and system RAM are being utilized) I went through multiple iterations of refactoring to implement more efficient algorithms and data structures. On the GPU side, I learned about and implemented several Nvidia specific enhancements, including Automatic Mixed Precision (AMP), to perform forward/backward passes in float16 where safe, and Triton, which enables PyTorch to generate custom GPU kernels that reduce overhead and potentially improve performance. Unfortunately, none of these efforts proved particularly successful. I was working locally on GeForce 3080 GPU and Ryzen 5600x CPU, and I don't believe I was constrained by hardware resources because during training runs, GPU compute & memory utilization were often under 20% (sometimes far less), and CPU utilization was generally under 40%. My only (unsatisfactory) conclusions at this time are that at either something is misconfigured on my system at a hardware/driver level, or if I were to continue working on this project, I would need to do a ground up refactoring with parallelization/multithreading as a top priority to much more thoroughly saturate the compute and memory resources available.

5. Experimentation & Results

For the standard DQN and DDQN using the CNN provided in the original assignment, I was able to successfully run controlled experiments of 750K training steps each. I used a pre-seeded replay buffer with 50K frames of gameplay from an earlier agent revision (to provide more initial diversity than starting from a completely randomized agent), a batch size of 64, epsilon annealed from 1.0 to 0.1 over the first 500K steps, the Adam optimizer, and a learning rate starting at .0001 that decayed 35% every 100K steps. The repeat action probability was set to 0 (i.e. completely deterministic behavior). This almost exactly mirrored the setup used in the homework assignment and the DQN paper. For both the DQN and DDQN, the trailing mean of episode score started to flatten out around episode 2250 [see fig 4]. I believe this may be due the training loop reaching its minimum epsilon, and starting to primarily exploit before it had learned how to successfully play deeper into rounds when less bricks remained and the ball is moving faster. Visual examination of gameplay videos saved periodically during training at least superficially supports this conclusion.

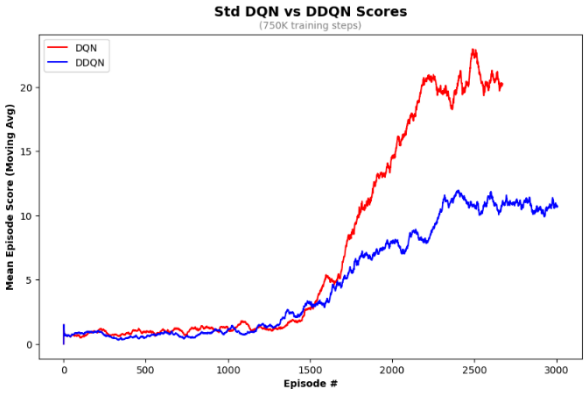

Figure 4: Trailing mean of the last 50-episode total scores (5 lives) over the course of 750K training steps
Note: num of training episodes are not equal because higher scoring episodes last longer result in more training steps per episode)

I also trained two additional models, the first a standard DQN agent with the previously discussed branched network architecture, and a standard DQN agent with a PER replay buffer and original network architecture. In both these cases, due to a variety of technical hiccups (crashes with imperfect checkpointing, etc.) combined with time constraints, the training runs were not able to be completed in as rigorously standardized manner as the first two tests. That said, each of these tests was still ~500-600K training steps. Hyperparameters were broadly the same as above, and for the PER test, I linearly annealed α from 0.6 to 0.2, and β from 0.4 to 0.8. Repeat-action-probability was also set to 0.05 in the PER test to explore the impact of removing determinism. Despite these latter two tests deficient experimental designs, results are nevertheless included here because I believe they are still broadly instructive.

Testing was done by using the trained agents to play 50 full games (5 lives each), using a consistent set of random seeds. To add some robustness, agents are set to use a ε-greedy strategy where ε=0.05 (i.e. 95% of the time actions are chosen using the trained model); this was the same testing methodology used in the DQN paper. Given the lack of any better standardized statistical measures (at least that I am aware of) to measure the accuracy or "goodness" of the trained RL agents/models, I relied on simply using the in-game score as a performance metric, much the same as was done in the DQN paper.

Model Description	Episode Score	
	Mean	StdDev
Std DQN, Basic Network	23.7	4.94
DDQN, Basic Network	10.2	2.80
Std DQN, Branched Network	15.3	3.63
Std, DQN w/ PER, Branched Network		

Figure 5: Testing results from 50 episodes of random-seed-controlled agent gameplay in *Breakout*

Qualitative assessment of these results is difficult, but in all cases performance of the agents exceeded the benchmarks defined as "success" by the Univ. Illinois assignment (8 for the standard DQN, 10 for the DDQN). I was somewhat surprised by how much worse the DDQN performed – the original DDQN paper did show lesser performance for *Breakout* specifically, but not to this degree. I suspect this can mostly be attributed to the more conservative Q-value estimates in the DDQN, and that if both models were trained over are much longer period (as they were in the original papers), this performance gap would close or disappear completely. As far as the two enhancements I tried, it's hard to say why performance fell below the basic DQN alone. Especially for the PER version, where there is published research proving out the method's efficacy, my assumptions is that there was either more training time was required, further optimization was necessary, or that there was an error in my implementation,.

Comparison against the results provided in the original DQN and DDQN papers is largely irrelevant for two main reasons. Firs, the number of training steps they used was an order of magnitude larger, and infeasible to replicate. Secondly, if I am interpreting the papers correctly, it appears their provided scores reflect running the agents for a fixed time/number of frames without controlling for the number of lives or game rounds. From an intuitive standpoint, I don't think this is the most logical way to judge agent "skill" – in *Breakout* the goal is to clear a many as bricks as possible in a single life/game. Although score does scale to a degree for clearing bricks higher up in the "wall", a relatively unskilled agent could still accumulate a relatively high score by playing many rounds poorly over a fixed time interval, as opposed to fewer, higher scoring rounds.

6. Takeaways and Possible Next Steps

Overall, I came away from this project with a much more in depth understanding about the intersection between deep learning and reinforcement learning. I gained a (sometimes painful) awareness of the fragility of using data generated as part of the reinforcement learning process as the training data for traditional deep learning. As discussed, I ran into multiple roadblocks that could not be overcome with brute GPU processing power/memory, as often seems to be the case in traditional deep learning (at least to come degree). Given more time, I would definitely like to determine the root cause of my suspiciously slow training time issues and determine definitively if a workaround could be found.

As a research-focused next step, I'd like to explore models that better capture the sequential structure of gameplay data. While I attempted to encode some temporal information using frame differences in a branched network, a more principled approach could involve using a CNN to extract per-frame feature embeddings, and then feeding these into a recurrent model — such as an RNN, LSTM, or even a Transformer — to learn temporal dependencies across time steps.

That said, it's not entirely clear how much this would improve performance in Breakout. If the state input to the agent is sufficiently rich (e.g., stacked frames or velocity-aware features), then the environment should satisfy the Markov property, meaning that the optimal action depends only on the current state, not on the full history. In that case, the benefits of sequence modeling might be limited.

Along those lines, during my initial research for this project, I came across actor-critic models, particularly Proximal Policy Optimization (PPO). While I didn't explore these in depth, my initial takeaway was that PPO is considered closer to the state-of-the-art in modern reinforcement learning due to its stability and performance in a variety of environments. Standard PPO implementations do not inherently leverage sequential data either, as they typically assume that the state input is already a sufficient (i.e., Markov) representation of the environment. It would be interesting to build an experiment to compare DQN vs PPO performance in Atari games.

Lastly, I recognize that the entire premise of training an agent to play a game using only visual input warrants further reflection. For certain types of environments — such as strategy board games — access to structured, logical representations of game states and rules would likely yield better performance than relying solely on raw visual data. However, in modern video games, where graphical fidelity is rapidly approaching photorealism, the image on screen is also the primary input modality for human players. In that context, one could argue that building agents to interpret and act on visual data is not only appropriate, but analogous to real-world applications like autonomous driving and other computer vision–driven domains. As the field evolves, I suspect that the boundaries between deep learning, reinforcement learning, and perception-based AI will continue to blur.

References

[1] Mnih, Volodymyr, et al. "Playing Atari with Deep Reinforcement Learning." *arXiv preprint arXiv:1312.5602* (2013).

[2] van Hasselt, Hado, et al. "Deep Reinforcement Learning with Double Q-learning." *arXiv preprint arXiv:1509.06461*

[3] Bellemare, M. G., et al. "The Arcade Learning Environment: An Evaluation Platform for General Agents." *Journal of Artificial Intelligence Research*, vol. 47, 2013, pp. 253–279.

[4] Lazebnik, Svetlana. "Assignment 5: Deep Reinforcement Learning." *CS 444: Deep Learning for Computer Vision*, University of Illinois at Urbana-Champaign, Spring 2024, https://slazebni.cs.illinois.edu/spring24/assignment5.html.

[5] Towers, Mark, et al. "Gymnasium: A Standard Interface for Reinforcement Learning Environments." https://gymnasium.farama.org (2024).