
A Comparative Study of DQN Performance in Atari Pong with Varying Environment Frame Rates

James Bardin ¹

¹Harvard University, Cambridge, Massachusetts

May 5, 2023

Question: Can Deep Q-Networks (DQN) trained on Pong Environments with specific frame rates generalize to perform well on environments with different frame rates, and, are different training frame rates better at generalizing?

1 Literature Review

Over the last decade there has been tremendous progress in the field of deep reinforcement learning, which has led to breakthroughs in industry automation, healthcare, natural language processing, and even gaming. With no prior knowledge, reinforcement learning agents are able to learn and master video game control strategies in times that

humans don't come close to matching. One of the leading works in this area, cited over ten thousand times, is the paper "Playing Atari with Deep Reinforcement Learning" by Mnih et al. (2013). The authors demonstrated the effectiveness of a DQN algorithm in playing classic Atari games, achieving superhuman performance on several different games without any adjustments to architecture or learning algorithm.

Since then, various modifications and extensions to the DQN algorithm have been proposed to improve its performance and stability. Deep reinforcement learning has significant challenge with computational requirements. The complexity of the algorithms are tough to reduce because of the inherent requirement for sequentiality. Training data is

generated during learning which seemingly leaves few options for significant speed and computational improvements. But, in "Reinforcement learning through asynchronous advantage actor-critic on a GPU" by Mnih et al. (2016) explores different possibilities. The main issue is that training batches are typically small and need to be efficiently guided to the DNN trainer. And, when using a GPU, small DNN architectures, small batches, both lead to sub-optimal utilization of computational resources. One modification they proposed was the asynchronous advantage actor-critic (A3C) algorithm, which enables parallel learning on a GPU and was found to achieve state-of-the-art results on several benchmark tasks.

More recently, the paper "Human-level control through deep reinforcement learning" also by Mnih et al. (2015) explored the potential that deep reinforcement learning algorithms have in solving more complex tasks, such as controlling a simulated humanoid robot, or again playing many different Atari 2600 games. The authors used a variant of the DQN algorithm called deep reinforcement learning with double Q-learning (DDQN) algorithm. And, they found they could achieve better performance and stability compared to the original DQN algorithm.

Each one of these papers highlight the leaps and bounds made in the field of deep reinforcement learning, as well as the potential these algorithms have to solve complex tasks, yet, reinforcement learning is a still blossoming field with countless discoveries and improvements left to be made. Some areas being focused on by research groups around the world include improving the sample efficiency, scalability, and generalizability of these algorithms to tasks that stray from the training situation.

2 Methodology

2.1 Experience Replay

The implementation contains several main components that work together to training an agent to solve Atari pong. First, we have our experience replay. The pseudocode for this section is as follows:

```
1. Initialize the replay buffer to be empty.
2. While the agent is not yet converged:
    1. Take a step in the environment.
    2. Store the experience (state, action, reward, next_state, done) in the replay buffer.
    3. Sample a minibatch of experiences from the replay buffer.
    4. Update Q function using the minibatch.
    5. Decay the epsilon value.
```

The experience buffer is a data structure that is used to maintain a history of past experiences. The experience buffer is accessed by the agent to store experiences as it takes steps through the environment. An experience, as shown in the pseudocode, contains the state, the action the agent took, the reward as a result of the action, the next state the agent is then in, and a boolean indicating whether the game has ended or not. The agent then takes a minibatch (a small subset) of experiences that are stored in the experience buffer using the defined experience data type, and uses these to update the Q-function with a method known as back propagation. The epsilon value we see in part 2-5 is another key concept. The epsilon is a hyperparameter that was carefully adjusted because it is responsible for controlling the exploration-exploitation trade off. When there is a high epsilon value, there are higher odds that the agent will make decisions—that according to what the agent knows—are suboptimal. And it does this with hopes of escaping local extrema because when the agent is still young

(hasn't observed significant data), it is likely to think that suboptimal strategies are optimal. So, as the epsilon decays, it takes fewer and fewer risky moves, more often sticking to what it has learned to work well. The experience buffer is implemented here **INSERT LINK HERE**.

2.2 Agent Class

Next the agent class was implemented. The agent class is used to determine the actions the agent takes inside of the environment we place it in. The agent class takes two parameters in its' constructor: the environment and the experience buffer. It contains several methods such as reset, which puts the agent's state back to the start and resets the total reward, used at the beginning of each game. It also has a play step function that takes an action according to the epsilon-greedy policy. The play step function takes a neural network, epsilon, and device as parameters. The neural network is what is used to estimate the Q-values of an action. The epsilon, determines the probability that the agent takes a risk step vs an well informed step. A random number is generated from 0 to 1 and if the epsilon is less than the number it chooses the action with the highest estimated Q-value from the neural network, and if it is less than the epsilon it will take a random action from the environment's action space. And finally, the device is where the neural network is being run on, so the CPU or the GPU. The agent class is implemented here **INSERT LINK HERE**.

2.3 Loss Function

Then, we have the implementation of the loss function. The loss function uses mean squared error (from torch.nn) to calculate loss. the function takes in a batch of data,

the neural network, the target network and the device. The batch of data, containing the same as the batches described before (state, actions, ...) are converted to PyTorch tensors and moved to the device for computation. The neural network is used to predict the Q-values for the state-action pairs. And the returned Q-value we treat as the expected sum of future rewards based on the action that was taken in the current state. Next, the target network is used to estimate the Q-value again, but now of the next state that has the highest Q-value. And this value is used to calculate the expected Q-value for the state-action pair by summing the immediate reward and future rewards. And finally, the expected Q-value is compared with the predicted from the mean square error function and returned. The loss function is implemented here **INSERT LINK HERE**.

2.4 Environment

The environment used for training is created from OpenAI's gym library. OpenAI has public wrappers for environments that make training an agent much easier. Examples of the wrappers implemented will start environments by pressing fire, apply various image preprocessing techniques such as data conversions, cropping, pixel reduction, and most importantly for this project, reduce the number of frames that are taken from the environment. We modified some parts of the wrappers from OpenAI and implemented the changes here **INSERT LINK HERE** in a file called wrappers skips.

2.5 Neural Networks

Then, the neural networks were implemented using the DQN class defined in the file named dqn model, which is a subclass of the torch.nn.Module class from PyTorch. The

model is a standard, simplified version of the model described in "Human-level control through deep reinforcement learning". It consists of a 4 layer CNN to extract features from the game state. And a fully connected network to predict Q-values for each action. The model is trained using a Q-learning algorithm, which updates Q-values for each state-action pair (s, a) ,

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Where alpha is the learning rate, r is the reward for some action a taken in a state s . The primes indicate the next state, and gamma is the discount factor. The DQN is trained using the experience buffer, which improves stability by preventing overfitting to training data. And, after training on a parameter grid, we found the most success using Adam's optimizer to optimize the network parameters.

2.6 Main Training Loop

So, that remains the is implementation the main training loop for the network. The network is trained by both using the experience buffer and the loss function. The target network is periodically updated (promoting stability). And, this training continues until the mean reward exceeds some predefined bound at which we consider the environment solved, or until it reaches a predefined number of games. The main loop follows the following pseudocode,

```
1. Get an action from the agent.
✓ 2. If the reward is none:
    1. Add reward to list of total rewards.
    2. Calculate mean reward.
    3. Save current mean reward and number of games in lists
    4. Print current information for user
    5. Stop if passed mean reward bound
3. If the replay buffer is not full continue
4. If the current frame is divisible by sync variable, then update the target network
5. Reset gradients of current network
6. Sample new batch from buffer
7. Calculate loss and backpropagate
8. Update parameters of network
9. Increment frame index
```

2.7 Conda Environment

This program was implemented inside a Conda environment with python 3.9.13. A yaml is provided in the Git repository called tfgpu env. This environment was designed for GPU acceleration support for Windows 10 with the help of Bex T.'s article published on Towards Data Science titled "How to Finally Install TensorFlow 2 GPU on Windows 10".

3 Results

Due to time constraints, many of the frame skip rates we only ran until we began to see significant decay in mean reward improvement rate, for somewhere around six hours on average. But, for five different frame rate values (skip every 3rd, 4th, 5th and 9th frames), we let the agent train for over 12 hours each. And, as a result we have well developed and highly performing agents in each of the environment types, and especially well performing models in the give environment types that were just specified. The plots for all skip rates from three to nine are shown below.

A Comparative Study of DQN Performance in Atari Pong with Varying Environment Frame Rates

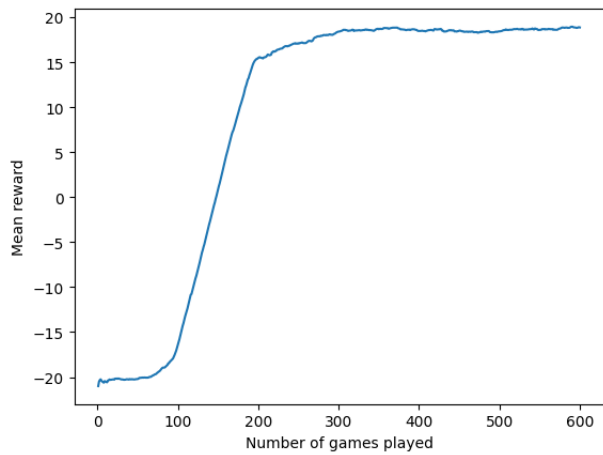


Figure 1: Mean Reward: Skip Every 3rd Frame

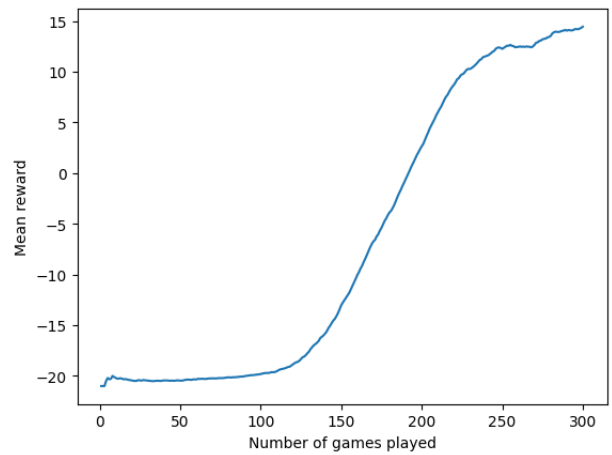


Figure 4: Mean Reward: Skip Every 6th Frame

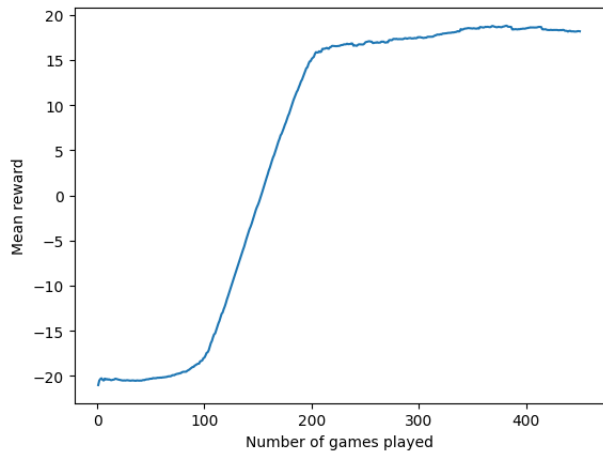


Figure 2: Mean Reward: Skip Every 4th Frame

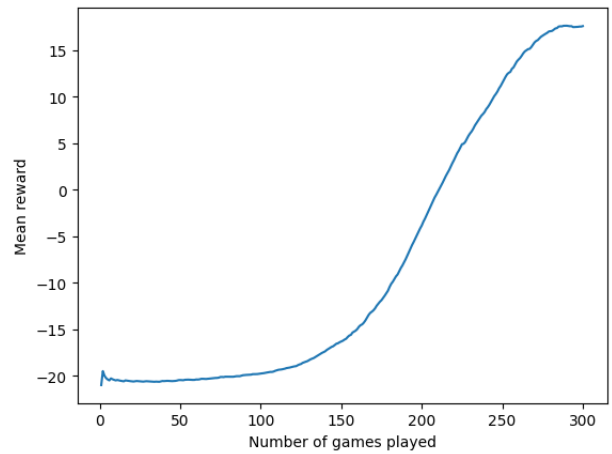


Figure 5: Mean Reward: Skip Every 7th Frame

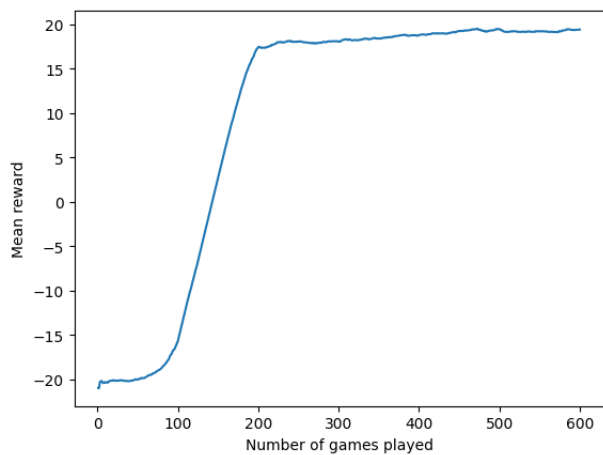


Figure 3: Mean Reward: Skip Every 5th Frame

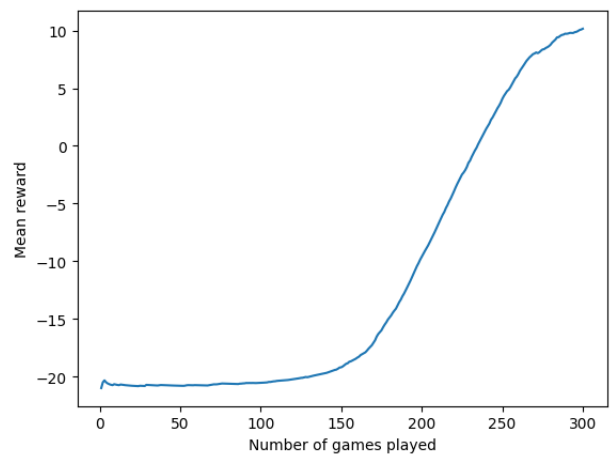


Figure 6: Mean Reward: Skip Every 8th Frame

A Comparative Study of DQN Performance in Atari Pong with Varying Environment Frame Rates

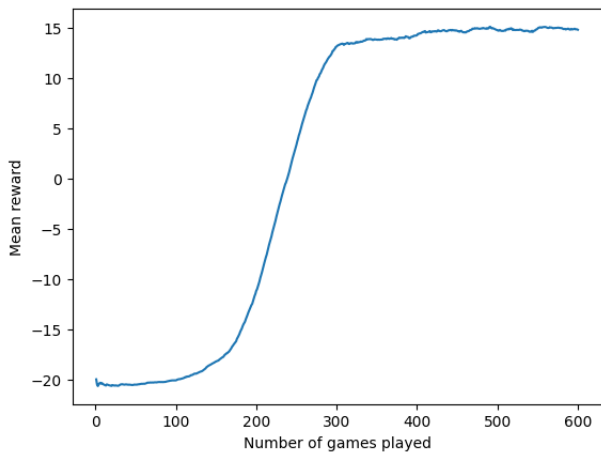


Figure 7: Mean Reward: Skip Every 9th Frame

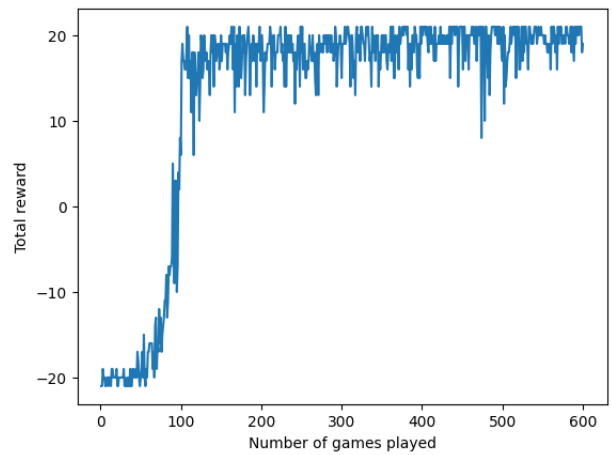


Figure 10: Total Reward: Skip Every 5th Frame

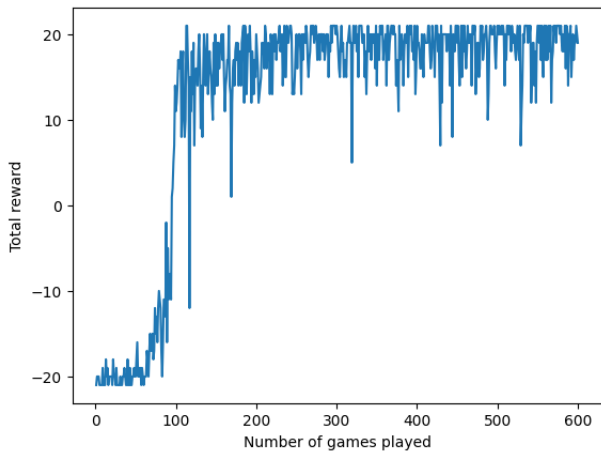


Figure 8: Total Reward: Skip Every 3rd Frame

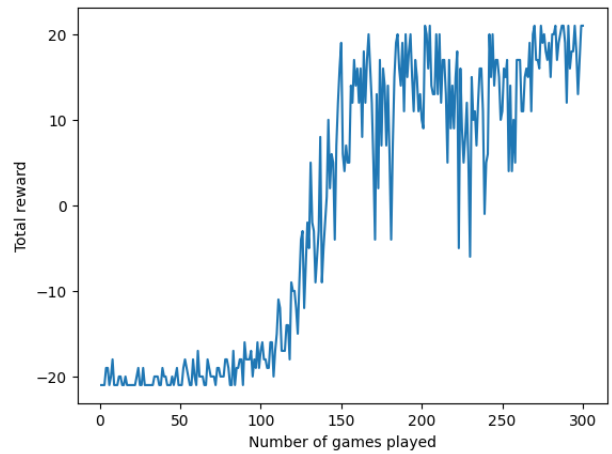


Figure 11: Total Reward: Skip Every 6th Frame

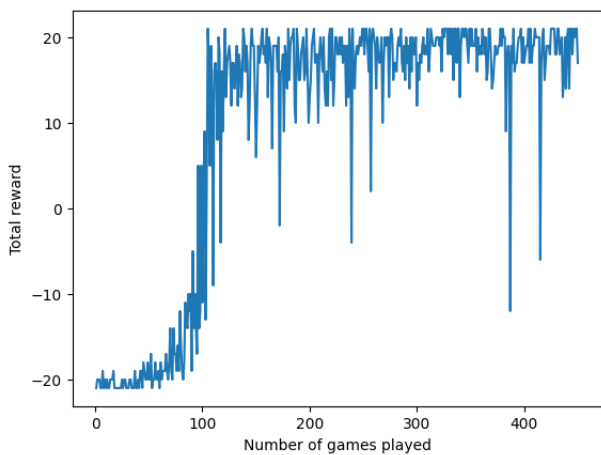


Figure 9: Total Reward: Skip Every 4th Frame

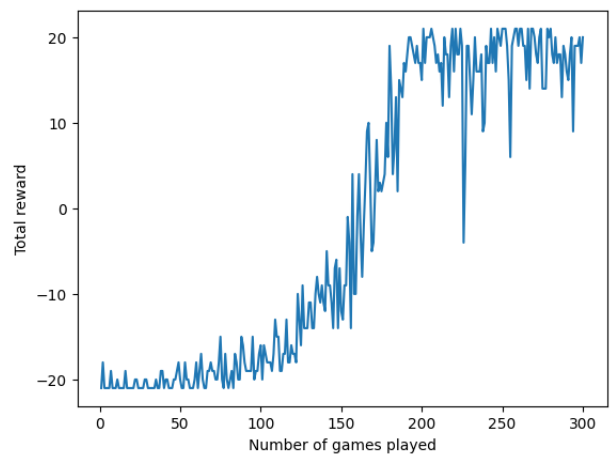


Figure 12: Total Reward: Skip Every 7th Frame

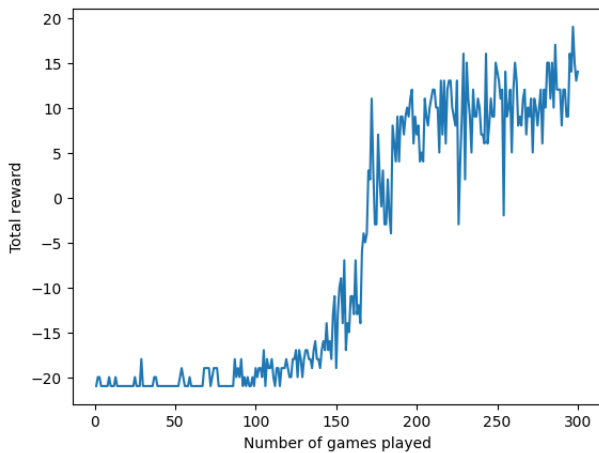


Figure 13: Total Reward: Skip Every 8th Frame

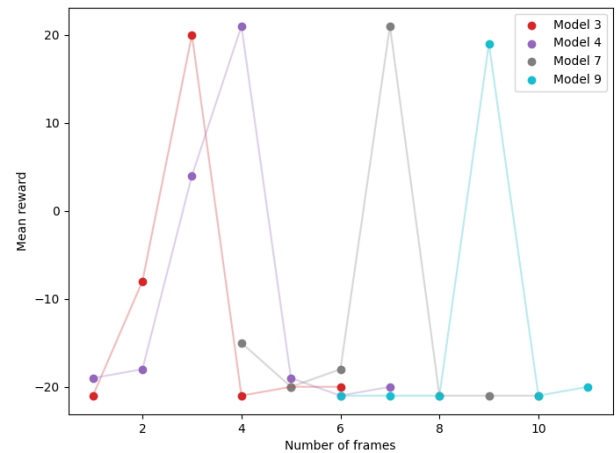


Figure 15: Performance on Varying Framerates

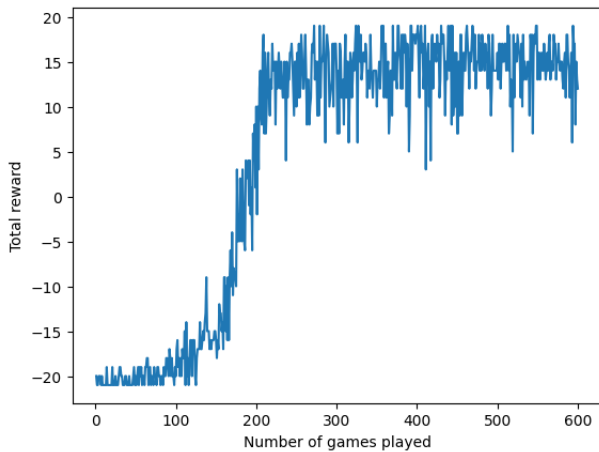


Figure 14: Total Reward: Skip Every 9th Frame

3.1 Further Examination and Discussion

After we had trained all of these models, until they were consistently beating the opponent handily, we examined the generalizability to different environment frame rates. We ran agents trained by models with frame skip rate of 3, 5, 7, and 9, on environments with values of frame skip rates within an integer value 2 from the rate. Each frame skip rate was run on each environment 20 times, and the average rewards were saved and plotted. We got the following,

Interestingly, the models were very poor at performing in environments with frame rates different from the environment they were tested in. When I first noticed this, I was curious if there was any performance improvements in models that skipped the same or different polarity frames, or where the frame skip was a multiple of the other one. For instance, would an environment that checks every third frame perform well in an environment checks every sixth frame or vice versa, and it turned out that none of these very able to generalize to other frame rates. This was interesting because it showed how important consistency in preprocessing is to model performance. The agents are trained on environments with specific game rates and thus learned to make decisions based on specific timings and speed of visual cues. Although there is not a perfectly clear explanation for this, one possibility is that when the agent is trained on a very high frame rate, the motion of the ball may not be perceived as linear. Because of the fairly small pixel space, the ball cannot move in a true straight line if it is moving at an angle different than 0, -45, or 45 (if zero is perpendicular to the paddle). In these cases the ball's path would be stepping like a staircase. And, in the case where the agent is trained on a lower frame rate, there

is more information loss. And, when in a different environment, the agent would have more trouble accurately perceiving the state of the game.

Upon this discovery, the question was raised whether the trained agent could adapt to varying initial seeds, since all environments were initialized with the same seed during training. The results showed that even when the environment seeding was randomized, the agent was capable of outperforming the opponent with the same average reward. And, interestingly the agent although beginning with a different sequence of volleys, would end up using the same strategy. After winning a point and the ball being served to the opponent, the agent uses the same strategy of bouncing a unreturnable ball off the floor or ceiling (depending on what is closest), always scoring after a single contact when the opponent is served. And interestingly, no matter what frame rate the agent was trained in, it always developed a similar strategy at playing the game.