A Single Q Network is All You Need
(A WORK IN PROGRESS)
Research Done by
Snir Vitrack Tamam
Ofir Ezrielev

Advisor: Ronen Brafman

March 9, 2022

Abstract

We present a new way to think about deep learning to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. We took DeepMind's architecture and modified the learning process in such a way we only needed a single Q network (SQN) instead of the usual fixed target network where we needed at least two.

Contents

1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long-standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation Silver [2013].

One solution is known as deep neural networks. Notably, recent advances in deep neural networks, in which several layers of nodes are used to build up progressively more abstract representations of the data, have made it possible for artificial neural networks to learn concepts such as object categories directly from raw sensory data Silver [2016].

However reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date have required large amounts of hand labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of time steps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution Silver [2013].

To address these problems, there are two common approaches: 1) A mechanism termed experience replay that randomizes over the data, thereby removing correlations in the observation sequence and smoothing over changes in the data distribution. 2) An iterative update that adjusts the action-values (Q) towards target values that are only periodically updated, thereby reducing correlations with the target Silver [2016].

We propose a novel way to maintain the stability, without compromising on the continuity of the training: We suggest to use a single model and updating it online but only updating a subset of the weights, we believe that in this way we will not ruin the stability and the model could learn and make decisions on the most recent updates.

2 Proposed Gradient Subset Selection Strategies

We considered a few ways to choose the gradients for the learning process.

2.1 Labels

Here we explain how to read the labels of our experiment: The left number corresponds to the maximum percentage of the non-zero gradients and the right number corresponds to the maximum fraction from the overall parameters. The amount of parameters we will update is the minimum from those two.

2.2 Random

The most simple way to approach this is to select the parameters to update in a completely random fashion.

2.3 Weighted Random

Choosing the parameters randomly but give more chance for gradients with bigger magnitude.

2.4 Greedy

Choosing the parameters with the most largest magnitude.

2.5 Epsilon Greedy

Choosing the parameters with the most largest magnitude but also take a random sample.

3 Pong

We tested our method on the game Pong with the gym environment. Pong is a table tennis-themed arcade video game featuring simple two-dimensional graphics, manufactured by Atari and originally released in 1972. In Pong, one player scores if the ball passes by the other player. An episode is over when one of the players reaches 21 points. In the OpenAI Gym framework version of Pong, the Agent is displayed on the right and the enemy on the left.

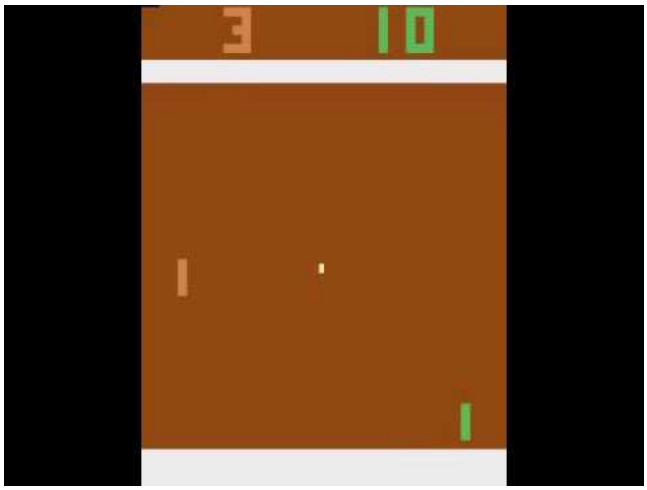


Figure 1: A frame from the game 'Pong'.

4 Code

Here we specify the classes of the project:

4.1 gym_wrappers

Atari games are displayed at a resolution of 210 by 160 pixels, with 128 possible colors for each pixel. This is a bit large and can be optimized. To reduce the complexity, we performed some processing:

- Converted the frames to gray-scale as color is not valuable for learning
- Scaled the image down to resolution of 84 by 84 pixels.
- Kept two subsequent observations frames as a single state, this helps the model to tell the direction of the ball.
- Skipped frames (showing only one out of four frames), this saves us time for learning since most frames are repetitive.

4.2 DQN

Is the class which contains the neural network, the network is made from convolution layers at the beginning and feed forward layers at the end similar to the architecture in Silver [2016].

4.3 ExperienceReplay

As suggested in Silver [2016] to collect training data in a buffer to remove the correlations of the updates, this class is responsible to collect the states in a tuple of state, action, reward and next state - (S, A, R, S). The buffer has a fixed size and once the buffer is full it will remove the most oldest experience. This class also contains a sample method where it samples a batch of experiences for the training phase.

4.4 PrioritizedExperienceReplay

This class is just like the ExperienceReplay but differs in two ways:

- It collects a tuple of state, action, reward, next state and the absolute TD error - (S, A, R, S, TD_err).
- When it samples it gives more chance to samples with bigger TD_err. This is done via tournament selection li [2010], where after k experiences were sampled uniformly the one with the largest TD error magnitude is chosen, and so on for the whole mini-batch.

4.5 Agent

Is the entity which interacts with the Environment, and saves the result of the interaction into the experience replay buffer. It selects an action using the epsilon-greedy behavior policy, and then saves the state, action, reward and next state in the replay buffer.

4.6 hook_handler

This script is performing the selection which sub-group of parameters will be updated during the training phase.

4.7 dqn_train

Is the loop where the training phase is happening for the regular DQN algorithm. The hyper-parameters are:

- Gamma, the discount factor.
- batch_size is the mini-batch size.
- learning_rate is the learning rate.
- replay_size is the replay buffer size (maximum number of experiences stored in replay memory)
- sync_target_frames indicates how frequently we sync model weights from the main DQN network to the target DQN network (how many frames in between syncing).
- replay_start_size the count of frames (experiences) to add to replay buffer before starting training.
- eps_start is the starting value of the epsilon.
- eps_decay is the amount of decay after each game.
- eps_min is thee minimal value of epsilon.

4.8 prioratized_dqn_train

Is just as dqn_train but instead of uniformly sampling with ExperienceReplay it will sample with more priority for large TD_err with PrioritizedExperienceReplay.

4.9 partially_dqn_train

Is the loop where the training phase is happening for our partially updated DQN algorithm. The hyper-parameters are as for the dqn_train but now we also need to choose Gradient Subset Selection Strategy.

4.10 prioratized_partially_dqn_train

The same as partially_dqn_train but instead of uniformly sampling with ExperienceReplay it will sample with more priority for large TD_err with PrioritizedExperienceReplay.

4.11 play_agent

Play and record a trained agent.

4.12 plotter

Generate the training graphs.

5 Experiment

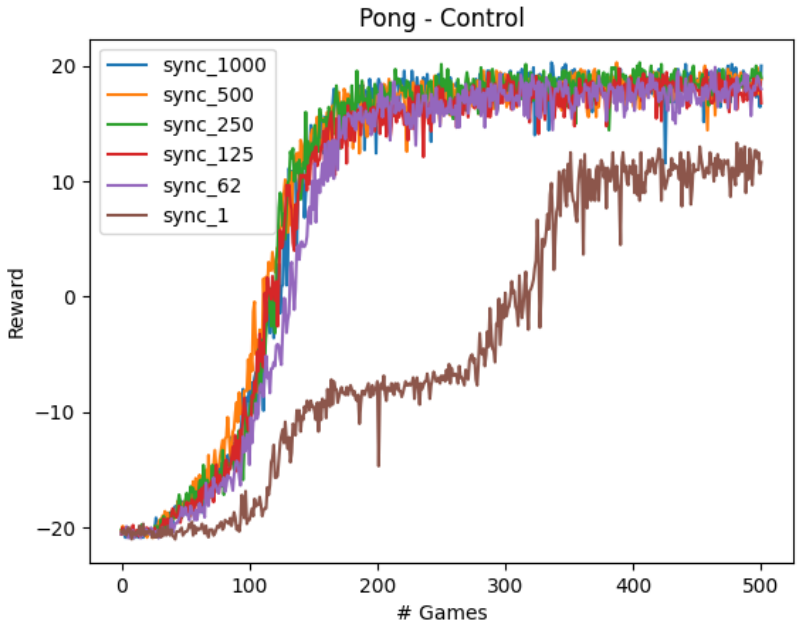
Due to the huge hyper-parameter space and the amount of time it takes to train a single model we had to select a specific set of hyper-parameters:

- Gamma = 0.99
- batch_size = 64
- learning_rate = 0.0001
- replay_size = 10000
- replay_start_size = 10000
- eps_start = 1
- eps_decay = 0.98
- eps_min = 0.02
- 500 games per model

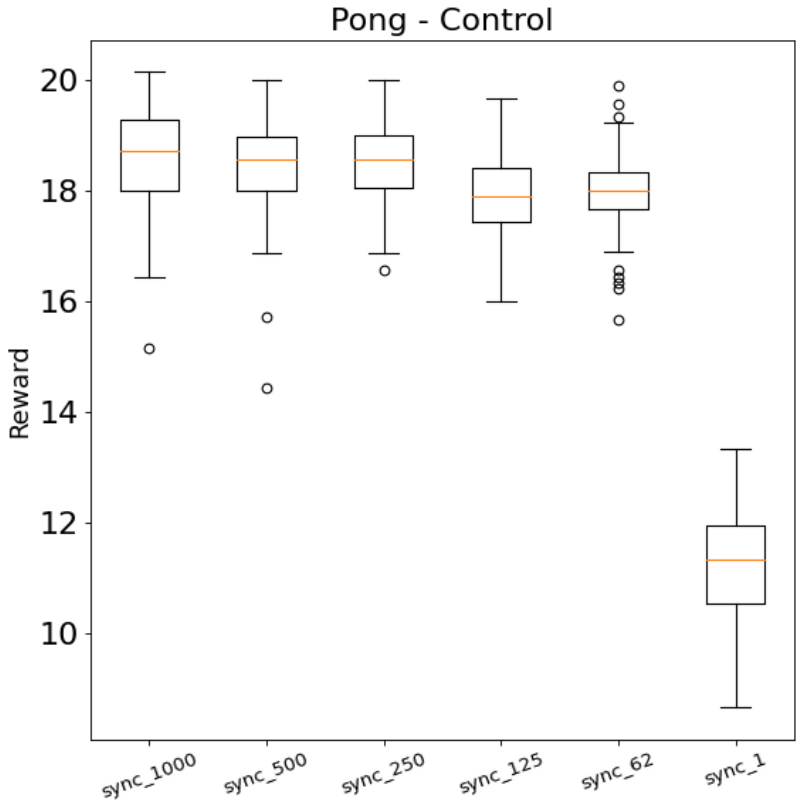
The DQN varied on the sync_target_frames with the values [1, 62, 125, 250. 500, 1000]

The SQN configuration varied with from how many non-zero gradients we can choose, values like 05_025 symbolize we can choose up to 50% of the non-zero gradients but not more than 25% from the overall number of parameters for the last layer, and we always choose the lower number. These values varied like [1_075, 1_05, 1_025, 075_075, 075_05, 075_025, 05_075, 05_05, 05_025, 025_075, 025_05, 025_025]

When more than a single layer is partially updated we just use this notation twice like 05_025_05_025.

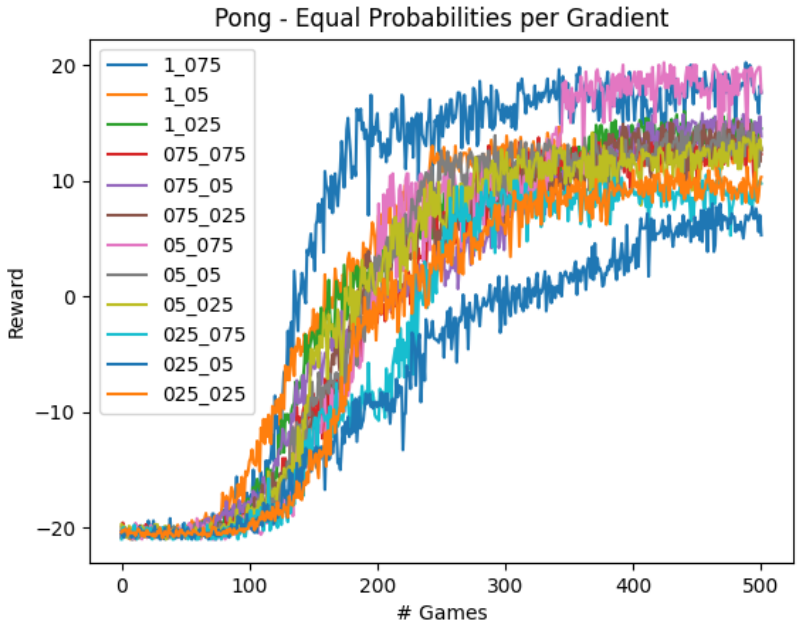


[a]

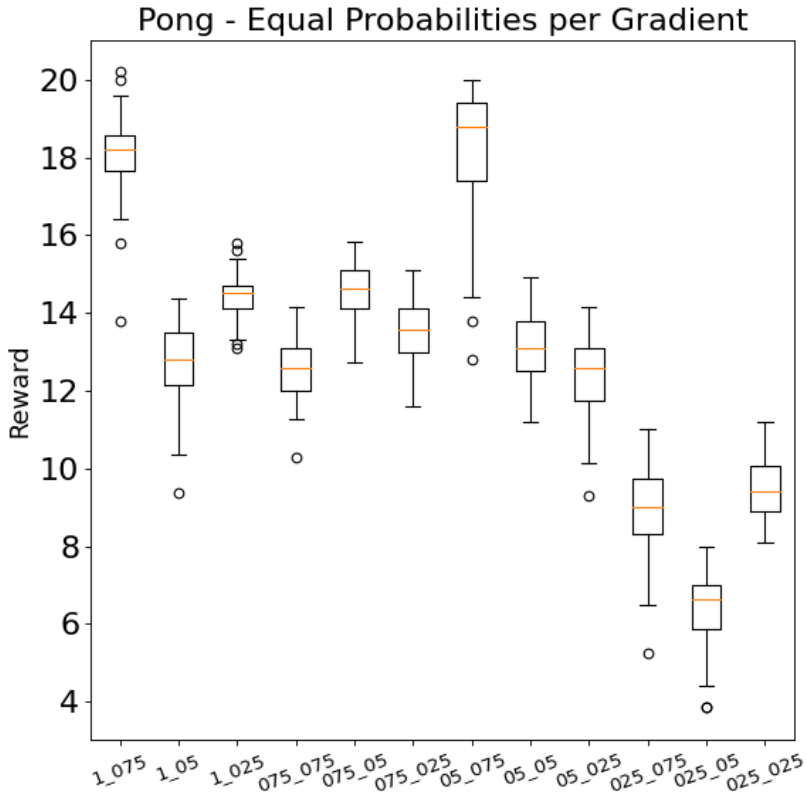


[b]

Figure 2: (a) Is the training process of the regular DQN algorithm where the numbering corresponds to the amount of frames before updating the target network. The X-axis is the amount of games played and the Y-axis is the reward. (b) Is the boxplot of the latest 50 games of each configuration.

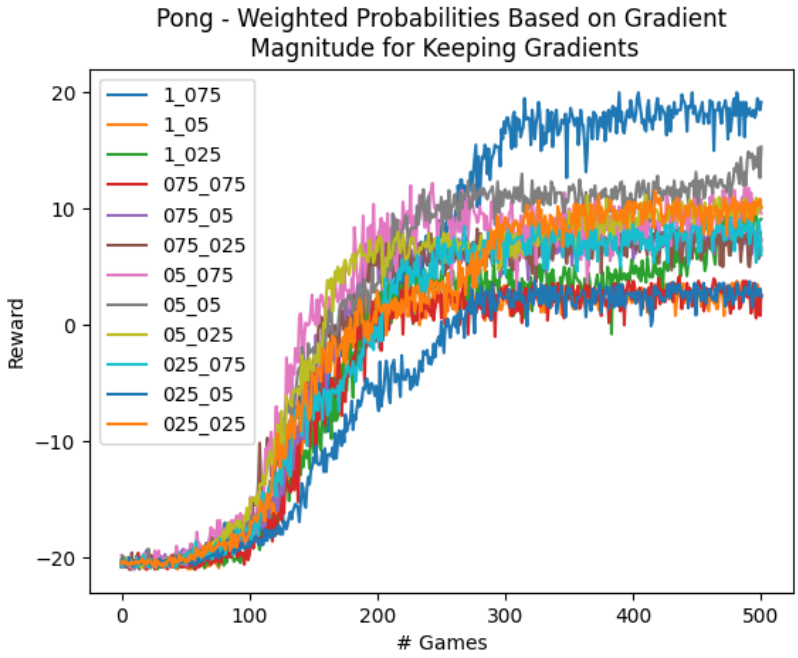


[a]

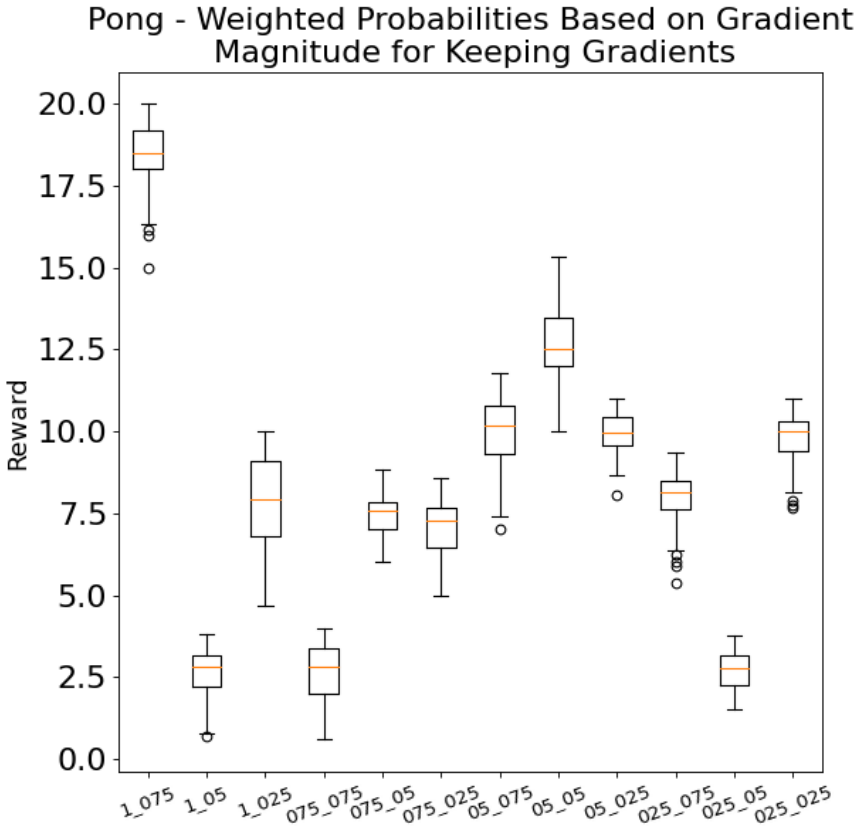


[b]

Figure 3: (a) Is the training process of the partially updated (in a completely random way) SQN algorithm. The X-axis is the amount of games played and the Y-axis is the reward. (b) Is the boxplot of the latest 50 games of each configuration.

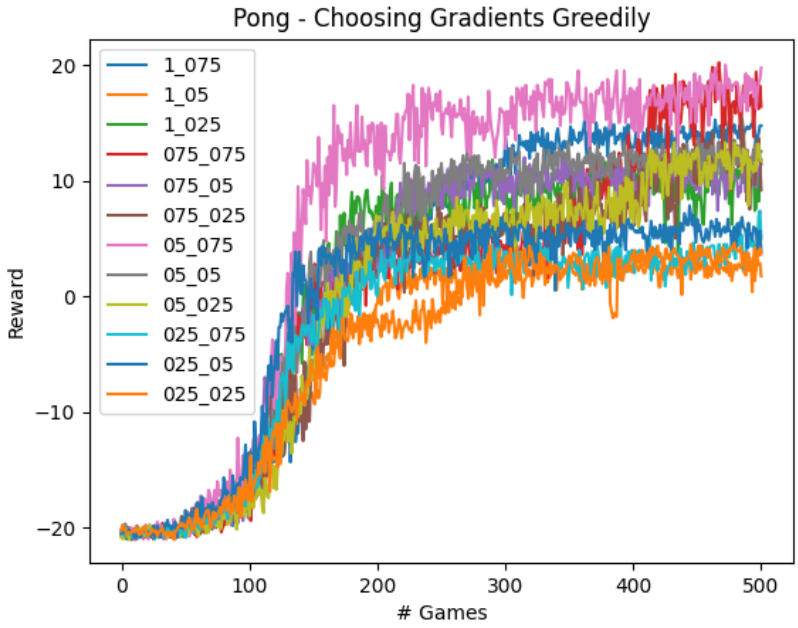


[a]

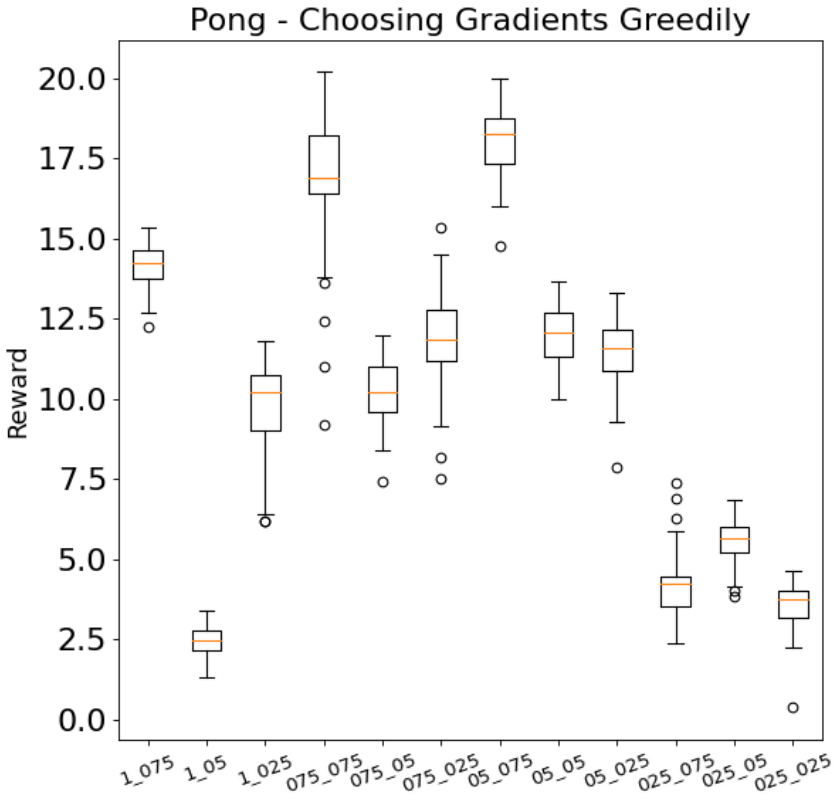


[b]

Figure 4: (a) Is the training process of the partially updated (in a weighted random way) SQN algorithm. The X-axis is the amount of games played and the Y-axis is the reward. (b) Is the boxplot of the latest 50 games of each configuration.

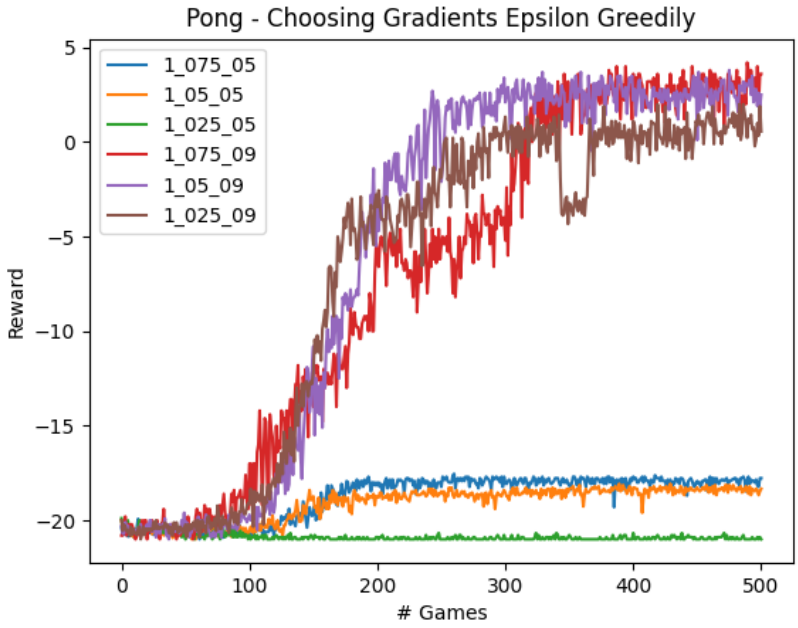


[a]

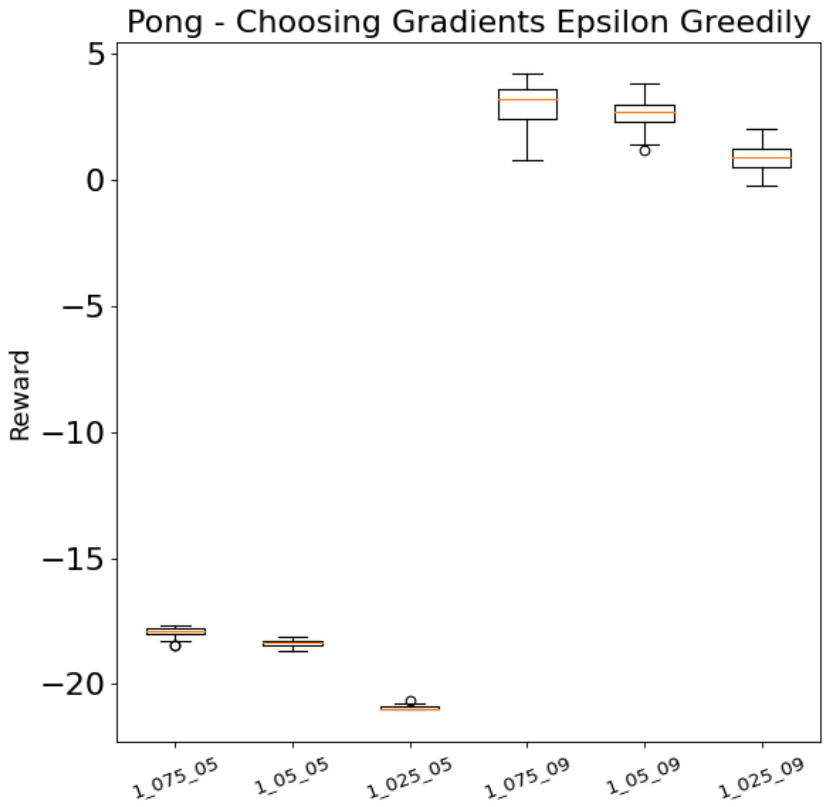


[b]

Figure 5: (a) Is the training process of the partially updated (in a completely greedy way) SQN algorithm. The X-axis is the amount of games played and the Y-axis is the reward. (b) Is the boxplot of the latest 50 games of each configuration.

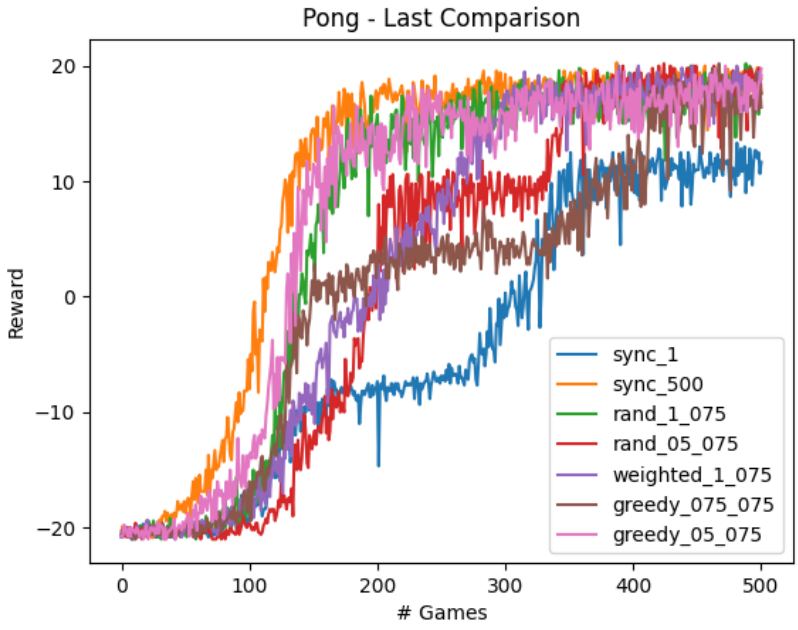


[a]

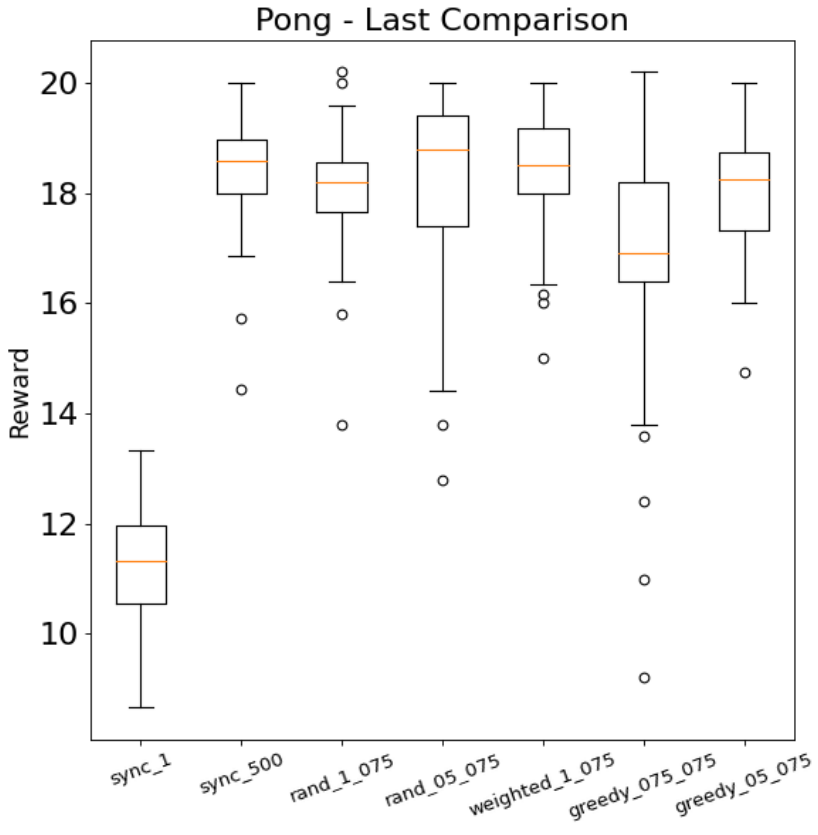


[b]

Figure 6: (a) Is the training process of the partially updated (in an epsilon greedy way) SQN algorithm. (The most right number corresponds to the epsilon). The X-axis is the amount of games played and the Y-axis is the reward. (b) Is the boxplot of the latest 50 games of each configuration.

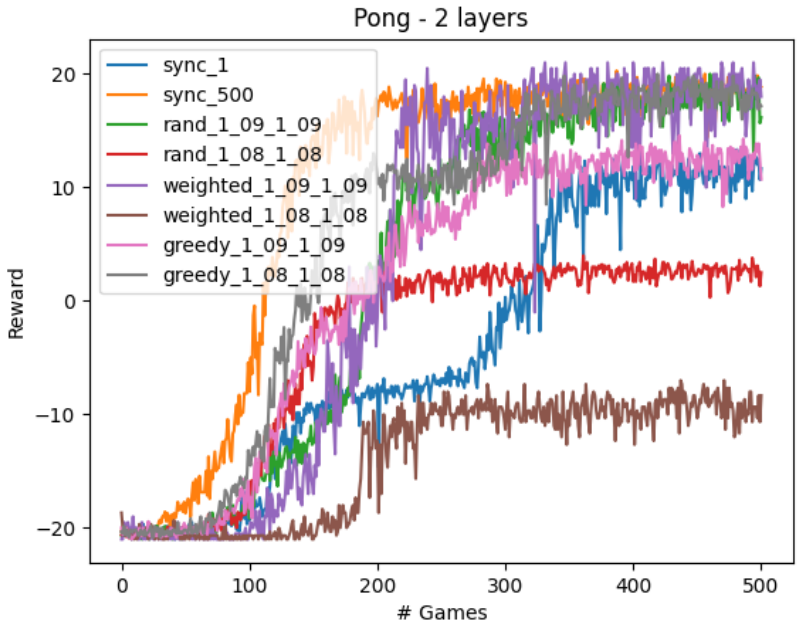


[a]

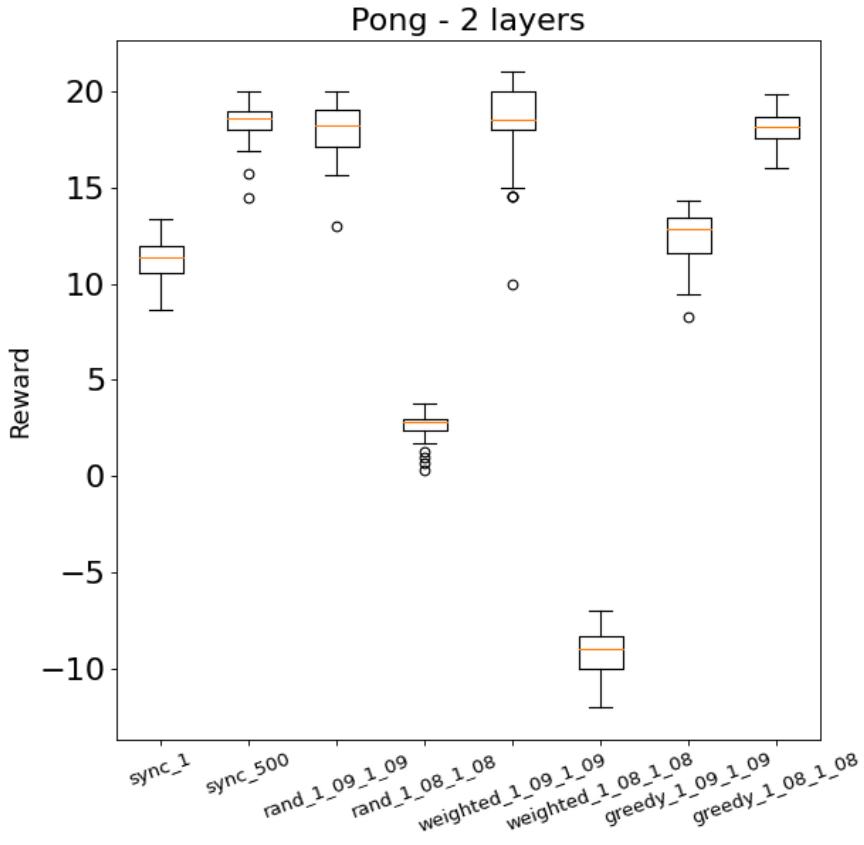


[b]

Figure 7: (a) Is the comparison of the top SQN updating algorithms compared to the regular DQN. The X-axis is the amount of games played and the Y-axis is the reward. (b) Is the boxplot of the latest 50 games of each configuration.

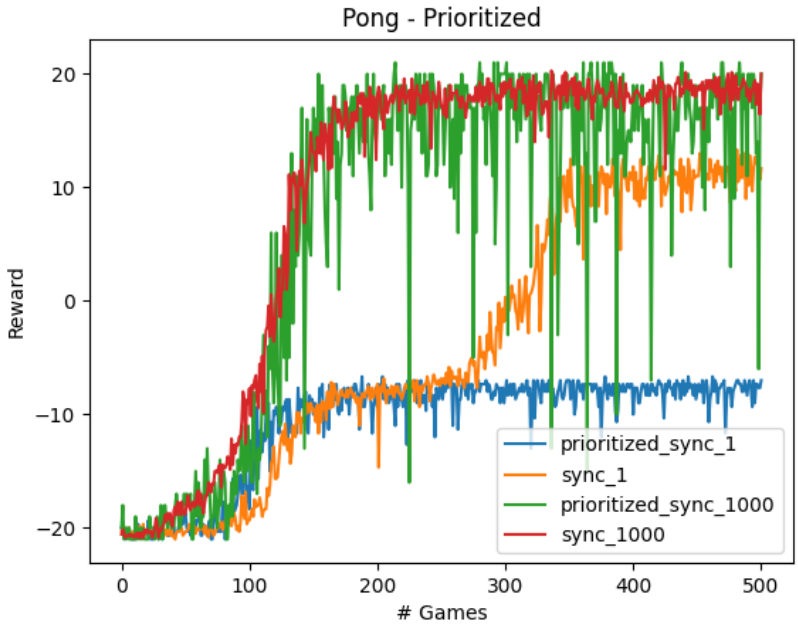


[a]

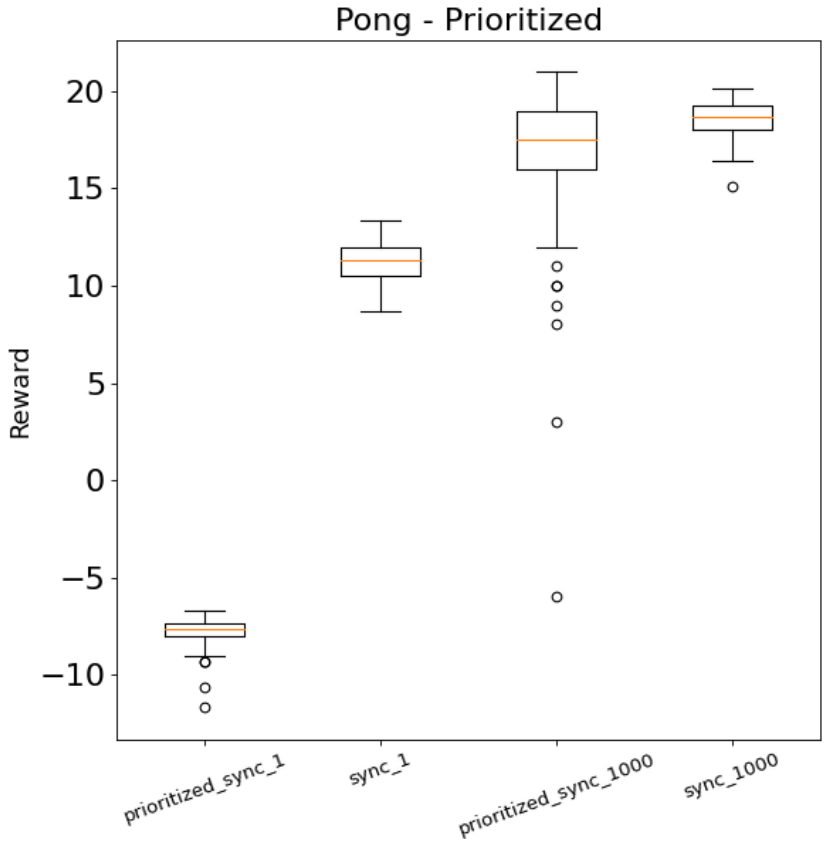


[b]

Figure 8: (a) Is the training process of the partially updated (in a various ways) SQN algorithm compared to the regular DQN. The X-axis is the amount of games played and the Y-axis is the reward. (b) Is the boxplot of the latest 50 games of each configuration.



[a]



[b]

Figure 9: (a) Is the training process of the prioritized DQN compared to the regular DQN. The X-axis is the amount of games played and the Y-axis is the reward. (b) Is the boxplot of the latest 50 games of each configuration.

6 Summary

Some of the SQN configuration came out as comparable to the regular DQN. It seems like most of gradients need to be preserved during the training in order to achieve competitive results to the DQN. It still seems like the our SQN is sacrificing convergence speed during training, but eventually converges similarly to the DQN.The user needs to decide if they can wait till the end of the training.This is still a work in progress and more configuration methods are being checked.

References

David Silver. Playing atari with deep reinforcement learning. 2013.

David Silver. Human-level control through deep reinforcement learning. 2016.

Jun li. A review of tournament selection in genetic programming. 2010.