Playing Pong with Deep Q-Network

Aron Sarmasi

June 12 2020

1 Extending the Deep Q-learner

In a deep Q-learner, a replay buffer holds a large number of state-action-reward-nextState tuples from the last N episodes. This is different than other supervised learning scenarios where the data-label pairs are all available from the start, because the learner in the reinforcement scenario must learn from continually updating data (ie. new interactions with the environment). One may wonder, why not simply perform a learning update after every single interaction, and eschew the buffer entirely? Although more memory efficient, this method is extremely prone to getting stuck in some local minimum. Intuitively, this method does not allow the optimizer sufficient foresight to optimize in the globally optimal direction. A replay buffer on the other hand holds many thousands of examples, and choosing randomly from it offers a balanced variety of state-action-reward-nextState tuples to learn from. A replay buffer is often combined with some flavor of temporal difference learning, where the rewards from two steps in the future of a stationary model are used to guide the predicted q values for one step of a continuously updated model. This can further reduce the tendency to get stuck in a local minimum.

Another important topic to discuss is that of random actions. The theoretical work backing Q-learning shows that if the state-action space is infinitely explored (every state-action pair looked at an infinite number of times), eventually the optimal policy will be learned. Since this is impractical, we take a more reasonable approach: every so often we pick a random action instead of the one with the highest q value. This way, we explore the consequences of actions which we may not already know to be good. As time goes on and we start to approach the optimal policy, it is no longer necessary to explore completely new/random possibilities as often, and building on the existing policy becomes a better use of our time. This is the reasoning behind the epsilon decay used in our deep Q-learner.

Before we get into parameter tuning, it is important that we discuss how the DQN was trained. Due to time constraints, the DQN was actually trained in two go's; both lasting 1M frames of play. This is visible in figure 1 as a large change in the average loss function at the 1M mark, and an increase in the slope of reward improvement–this last bit is due to a doubling of the learning rate at that point. There is not a principled reason for why we trained the DQN in 2 go's; it simply happened to be the case that there was a partially pretrained model from an earlier run ready to go, and we used this to save time during our hyperparameter search. The results of the search showed that a gamma of 0.995, a learning rate of 2e-5, a replay buffer of size 200,000, a copy frequency of every 40,000 iterations, and an epsilon decay of 100,000 yielded agreeable results. Since 5 hyperparameters are difficult to search over simultaneously, we simply did a random search with 7 random picks of different values, and this worked best. A few important takeaways from our search is that setting

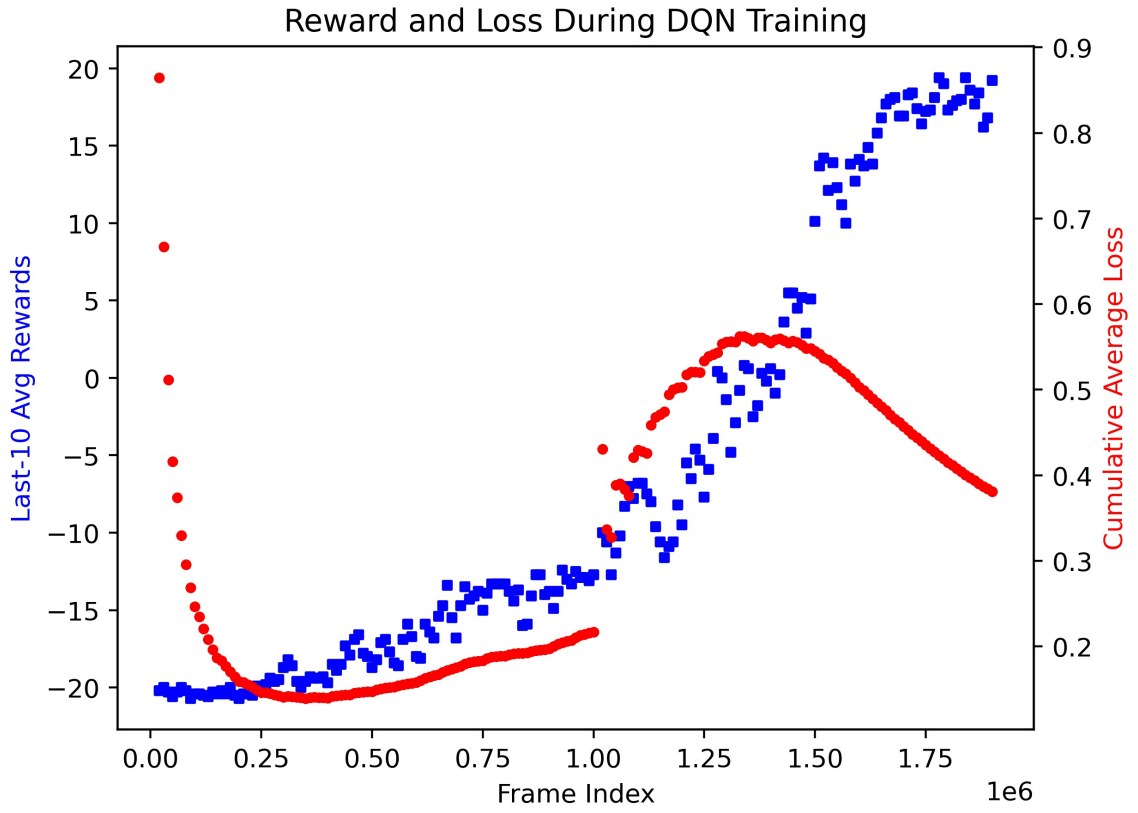

Figure 1: Due to the piecewise training of the DQN Pong agent, the rate of reward improvement doubles around the 1M mark as the learning rate is doubled. The average loss also jumps because epsilon is reset to 1. Interestingly it does not drop for a while, likely because the epsilon decay is triple of what it was in the first half. The highest last-10 reward achieved was 19.4.

the learning rate too high causes divergence, and that even the perfect set of hyperparameters isn't a substitute for patience; it simply takes a large number of frames to train the DQN agent.

2 Visualizing and Analyzing the Trained DQN

In this section we apply a clever manipulation to the board state representation so that a meaningful distance between two board states can easily be calculated. The manipulation consists of 3 energy maps, one for the ball, and one for each paddle. The energy value of each pixel in each energy map is simply the negative Euclidean distance from the nearest object of interest. This way, when comparing two board states, the mean squared error of their energy maps will be linearly (or at least monotonically) correlated with the distance between the various objects of interest. This is as opposed to performing the mean squared error directly on the board states themselves, which would result in no difference between two states that are nearly identical (ie. the game pieces are adjacent) or totally dissimilar.

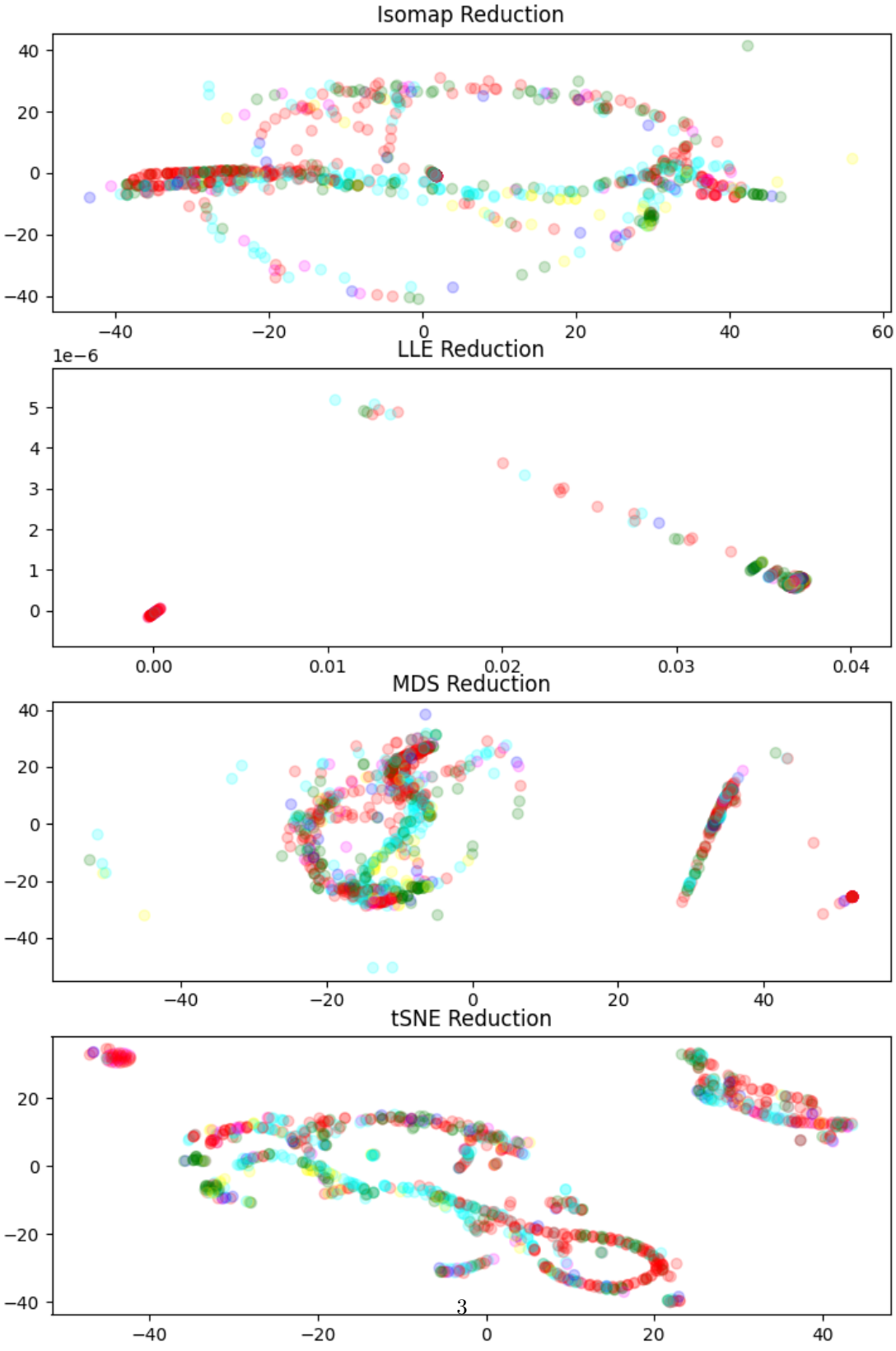

Figure 2: We show four different methods of dimensionality reduction. Although they are all fairly noisy, the color red – corresponding to the NOOP action dominates, and often forms clusters.

Using this manipulation, we are able to easily and meaningfully calculate distances and similarities between board states. Once we are able to do this, it becomes trivial to use an existing dimensionality reduction package; indeed we demonstrate dimensionality reduction with 4 such packages: isomap, locally linear embedding, multi-dimensional scaling, and tSNE. Please observe the results in figure 2.

However, such dimensionality reduction is almost unnecesary. Sadly, it is not possible to include a video in this document, so we will instead describe the actions of the agent. The policy that is learned hinges on the fact that the AI pong player is deterministic. Thus, the DQN makes the exact same set of moves each time, and the AI responds with the exact same set of responses. The DQN managed to find a winning set of actions this way, and it follows this exact same set of actions every time, all 21 times (with some mistakes) to achieve an average reward of 19.4. It starts off stationary in the corner, as shown in figure 3, and then erratically moves to the other side where it expects the ball to be (figure 4), performs a quick motion sending the ball to a preplanned position where the opponent is sure to miss it, and then it goes back into its corner.

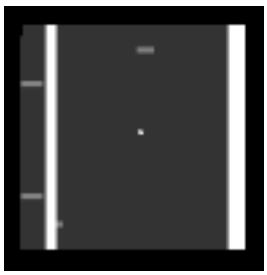

Figure 3: The agent (bottom paddle) waits until the opponent hits the ball; it does so in the same way every time. This corresponds to the red regions in the dimensionality reduction graphs above.

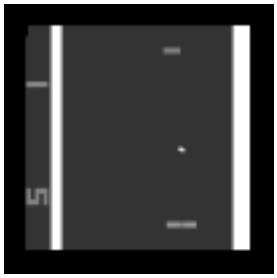

Figure 4: When the ball approaches on the lower right, the agent always does the same left-right jig, and always sends the ball on the exact same winning trajectory. This roughly corresponds to the green and cyan areas in figure 2.