Hyperparameter Optimization for Deep Q Networks: Project Report

Collaborators: *Ananth Ravi Kumar, In Wai Cheong*

Project Description:

To measure the sensitivity of Deep Q-Networks on different tasks subject to learning rate, batch size, optimizer, target Q network update step size, discount factor, and other hyperparameters to identify the relationship between hyperparameters and efficient convergence to the optimal policy across different state/action regimes.

Problem Motivation:

Since hyperparameter optimization for Deep Q Networks is an ill studied and computationally expensive task, we wanted to see if we could apply hyperparameter tuning methods that have shown promising results in Machine Learning and Deep Learning to Deep Reinforcement Learning using simple games and architectures. Furthermore, we wanted to see if there was a relationship between difficulty of tasks (in terms of combinatorial size of state/action space) and sensitivity to hyperparameters. The three tuning methods we used were random search, successive halving, and bayesian optimization.

Background Work:

In this section we will be discussing the details of the hyperparameter optimization methods. The first method we will discuss is random search. Under this method, hyperparameters are selected randomly within pre-specified constraint ranges. This method has been shown to be far more efficient than grid search in high dimensional search spaces for a multitude of reasons - it is intrinsically parallelizable (trials are iid), allows for different hyperparameters to have different importances to the downstream task, random search can be stopped anytime, and additional experiments can be added ad-hoc as resources free up when working within a distributed setting.

Successive halving is an evolutionary approach with random initialization.Under this approach, we randomly initialize a number of hyperparameters from the hyperparameter space, and evaluate their performance on the downstream task. At each stage we discard the bottom-half performing sets and keep the top-half performing ones. We repeat this until one set of hyperparameters remain. The assumption made here is that good hyperparameter sets will display benefits early on in training, but we might not end up with a good set of hyperparameters if the original set initialized were all poorly performing hyperparameters.

Bayesian Optimization attempts to utilize past information in the following manner. The main issue with grid search and random selection is that subsequent runs are not dependent in any way on the actual performance under those hyperparameters - as a result, a lot of time is wasted searching regions of the hyperparameter space that all lead to poor results. Bayesian optimization attempts to resolve this issue by trying to construct a distribution $p(x|y)$, where x is the hyperparameter vector, and y is the reward obtained on the task under those hyperparameters. By maximizing this probability, it effectively narrows in on regions within the hyperparameter space that provide the best results. The practical implementation of Bayesian Optimization that we utilized is called the Tree Parzen Estimator (TPE). This estimator takes this idea further by simultaneously learning a threshold y^*, and then learning two different probability distributions $l(x|y)$ and $h(x|y)$ where the former is sampled for $y > y^*$, and the latter is sampled for $y < y^*$. By partitioning the search space in this fashion, it allows for a more complete probabilistic description of the hyperparameter space, potentially yielding better results.

Approach:

Let's discuss a bit more of our workflow. We used OpenAI gym and tested on four games: MountainCar-v1, CartPole-v0, LunarLander-v2, and Acrobot. We decided to use discrete-action spaces due to the fact that Q-learning is ill-defined on continuous action spaces. Given that the Bellman equation requires taking an argmax over all possible actions a that can be taken from a state s, DQNs cannot handle continuous action spaces.

Our network architecture is a simple fully-connected network with 1 hidden layer of 100 neurons, utilizing a LeakyReLU activation. We decided to use a simple architecture to constrain the expressiveness of the network itself, so we could be sure the success of our model was due to good optimization of hyperparameters and not the architecture. The simple network also had the added benefit of reducing the hyperparameter search space and reducing training times.

Implementation Details:

We used the Optuna MLpPolicy and Stable Baselines libraries for the Successive Halving implementations, and Hyperopt's Tree Parzen Estimator (TPE) for Bayesian Optimization. Our input was the dimensions of the state space and our output was the dimensions of the action space. The hyperparameters we considered were the optimizers (we used Adam, RMSProp, SGD, and AdaGrad), learning rate, target Q update count, batch size, epsilon/epsilon decay, and gamma (reward discount factor).

Experiment Design Flow:

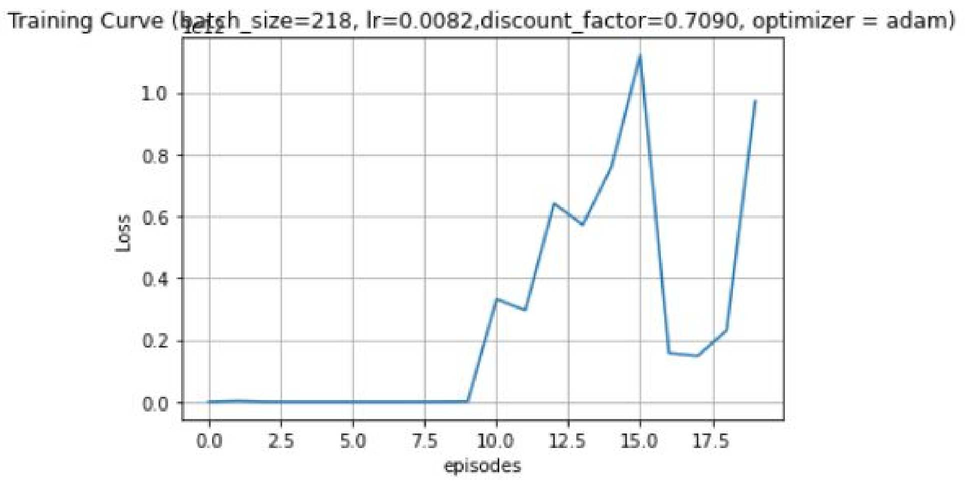
For random search, hyperparameters were randomly selected from a given range. For SH, we used Optuna & SB and we compared performance of hyperparameter combinations obtained to display sensitivity of RL convergence to hyperparameters. Using hyperopt to implement Bayesian Optimization, optimal parameters were selected through 1000 rounds of evaluation. A new model was instantiated and trained with these hyperparameters and the behavior of the initial model and optimal model is compared.

Results & Evaluation:

Random Search:

Random Search was by far the worst performing tuning method, as visualization of the agent in LunarLander just shows it crashing even when we used very high iterations. One interesting thing we noticed was that even if the loss plots were very well-behaved for a given set of hyperparameters, the agent could also not be learning anything. That is to say, the DQN obtaining low loss values did not necessarily translate into good performance on the actual game.

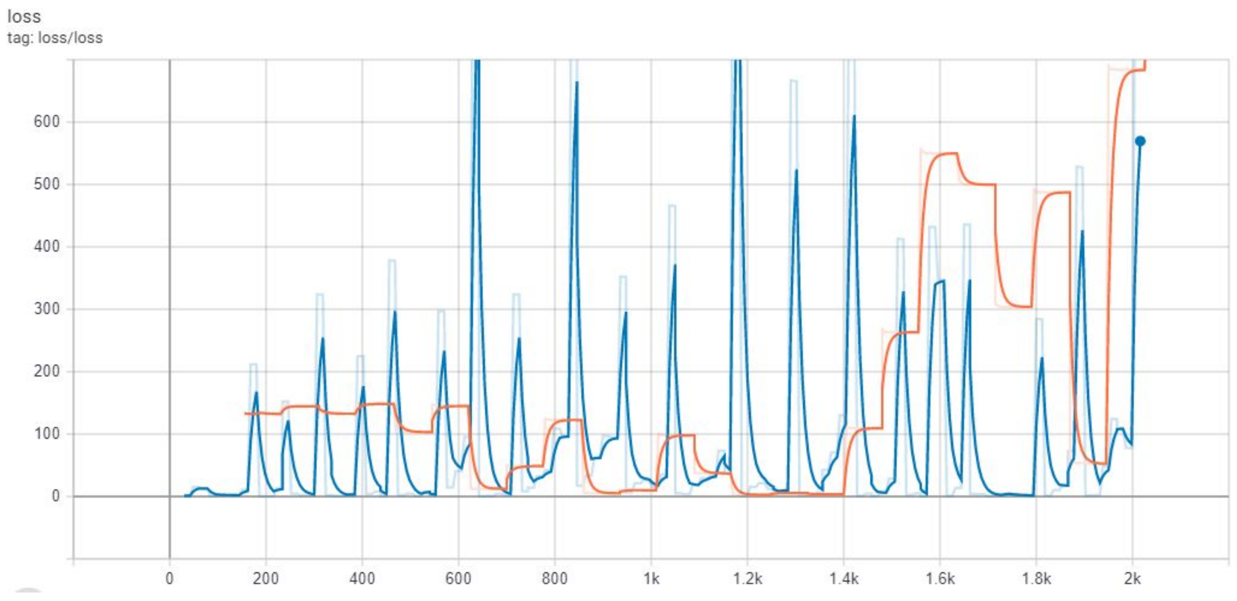
For random search, the episodes vs. rewards plot was -1 across the board; however, given a high enough number of iterations, it sometimes achieved the goal merely by taking random actions, and irrespective of the hyperparameters actually chosen. An example of what most of the loss plots look like for all four games is below:



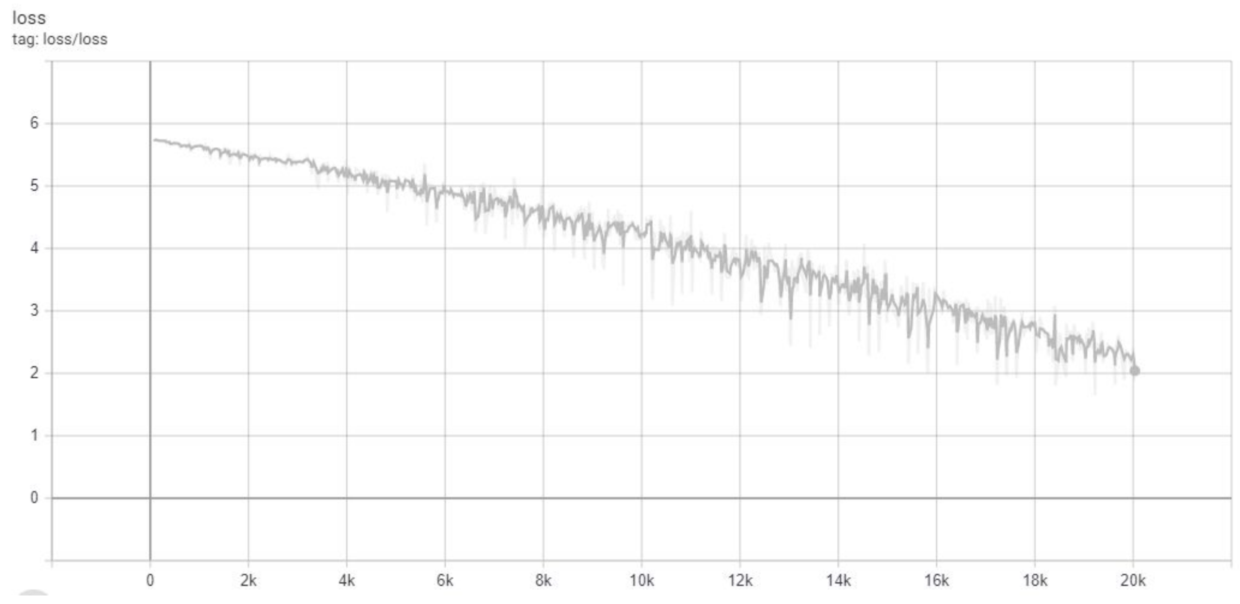
We did experiment with manually setting the hyperparameters but it performed just barely better than random search & is very dependent on the range selection of your hyperparameters.

<u>Successive Halving:</u>

Successive Halving definitely had its share of bad examples, one of which is displayed below:



The blue and orange lines both correspond to two different sets of hyperparameters that SH returned, and as mentioned before - this could be due to the initialization of the parameters in the first stage which were all poorly performing. But as we increased the number of trials for SH to repeat, for all four games we do end up with a set of hyperparameters who loss plots look relatively nice:



For LunarLander using SH and 150k iterations, we ended up having our LunarLander perform pretty well, shown here. We attempted to use hyperband, which we believe will do better than SH, but we were having trouble working with the library due to architectural problems. However, it is very much something we will be exploring in further projects.

Bayesian Optimization:

Bayesian optimization had some notably poor results. To reiterate the training loop, we first ran HyperOpt's bayesian optimization algorithm for 100 iterations to arrive at a set of supposedly optimal hyperparameters. A new model was then instantiated with this very set, and was re-trained on the same task. The behavior of the agent pre- and post- optimization was then compared to determine if any substantial improvements were achieved by virtue of the hyperparameter choices.

Disappointingly, Bayesian optimization yielded very substandard results, not at all worthy of the compute time required to run the optimization algorithm itself. There are a few potential explanations for this behavior. First, it might have been the case that the hyperparameter space as we defined it is not amenable to parametric estimation. Some of the hyperparameters that we were optimizing over were categorical (i.e. the optimizer), and it could very well be the case that these variables throw off the estimator.

Secondly, as described in the presentation, Deep RL presents us with the unique situation where we have dependencies *between* hyperparameters. For example, the batch size and the size of the experience replay buffer are clearly dependent on one another, in that having a batch size larger than the buffer length would be pointless. Additionally, the target Q network update rate and the size of the memory buffer also are dependent, because the classical DQN algorithm has the agent take random actions until the memory buffer is at capacity. It might be that these dependencies are not explicitly accounted for by Bayesian optimization, or even if they are, that the overall sparseness of the reward signal in these games prevented their discovery.

Technical Challenges:

We ran into a multitude of challenges while attempting this project, which we have roughly categorized as either conceptual or system-related challenges.

Conceptual Challenges:

The most glaring challenge we faced was how to go about assessing the correctness or the 'goodness' of a policy selected by an agent. As mentioned before, we encountered many scenarios where the loss from the DQNs was close to 0, but where the agent failed to complete the task. Relying solely on reward vs iteration plots also suffered from this drawback, in that our agent's solutions were often so random that they would almost accidentally succeed at least once. Our solution was to rely on direct visual inspection - which as we will see in the next section, came with its own set of problems.

We were also unprepared for just how long the training times of these agents would be, and for the sample complexity of the training. In fact, our disappointment in our agents' performance after 100,000 iterations led us down a rabbit hole of debugging our DQN implementation - it wasn't until late in the project that we realized that most Github repos attempting similar tasks often trained their agents for *millions* of iterations.

System/Resource Issues

OpenAI, with its reliance on visualizations and physics engines, does not play well with Colab or with GCP. This led to many compute-intensive runs crashing completely during the visualization phase, after spending dozens of hours training. What made this additionally frustrating was that Bayesian Optimization is not at all amenable to parallelism, unlike successive halving and random search - the poor results it obtained were even harder to understand given the massive compute cost.

Conclusion:

Our conclusions are as follows. First, that the sample complexity of DQNs is very very high even for networks and tasks as simple as ours - in our view, improving the sample complexity of DQNs would be an interesting research area. Secondly, that successive halving is the best performing out of the three methods we discussed. This is probably due to its nonparametric nature, and the fact that it can be easily parallelized, giving the most bang for its buck in terms of compute time.

Additional areas that we would have liked to explore given more time are listed below:

1. Hyperparameter importances for different kinds of games - could it be the case that some hyperparameters are more important than others for certain games, and that this is evident across multiple games of the same type?

2. Hyperband-based optimization: this was initially on our list of methods, but we found that adapting available implementations was a process riddled with errors and bugs.

3. Continuous-action spaces. Discretizing continuous action spaces would have presented us with an entirely new set of hyperparameters, i.e. the manner in which this discretization was done. It would have been interesting to evaluate the behavior of the various optimization schemes as they scaled with state/action size.

References:

1. Random Search for Hyperparameter Optimization: Bergstra and Bengio, 2012. https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a.pdf
2. Algorithms for Hyper-parameter Optimization: Bergstra et al, 2011 Algorithms for Hyper-Parameter Optimization (nips.cc)
3. Hyperband: A Novel Bandit-based Approach to Hyperparameter Optimization: Li et al, 2018. [1603.06560] Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization (arxiv.org)