Comparison and Best Practice for Two Continuous Q-Learning Methods

James Watson
dept. of Computer Science
University of Colorado Boulder
Boulder, USA
james.watson-2@colorado.edu

***Abstract*—Q-Learning [1] is a simple, yet relevant, reinforcement learning method that approximates the value function of performing an action in a state with a lookup table. Its primary drawback is that it is a discrete method in its original formulation. There have been many attempts to extend the utility of this method to continuous spaces. In this article, we explore the application of two of these methods to a continuous control problem and document the impact of varying the methods' hyperparameters on control performance.**

***Index Terms*—reinforcement learning, model based, Q-Learning, continuous state and action**

I. Background & Related Work

The methods studied in this work both rely on interpolation to essentially convert traditional Q-learning from a discrete method into continuous methods. There is basis for this early on in the history of Q-learning, as shown by Horiuchi *et al* [2]. Also like [3], the authors base their interpolation on membership in fuzzy sets. In both contexts, degrees membership in sets are used as weights for a linear combination of input values. Horiuchi *et al* the parameters of a linear value function by ascending the gradient of the temporal difference error with respect to those parameters. The main disadvantage of the method is that the fuzzy sets are hand-designed by a human expert, rather than being on a regular grid, as they are in [3].

Emigh *et al* [4], also exploit interpolation kernels in their method. The authors use kernels for interpolation, which is advantageous because experience from early, sparse samples can still be exploited. The difference interesting aspect of their approach to kernel regression is that their kernel is not fixed. In metric learning, the weights of an importance metric $[\sum_i \omega_i(x_i - x'_i)]^{0.5}$ are learned such that the features which contribute the most to changes in $Q(x, a)$ contribute the most to the difference metric between two states. This approach avoids the tricky work of feature engineering entirely by allowing the system to change the actual metric space to solve the problem, and helps to overcome the "curse of dimensionality" by simply attenuating the impact of unimportant dimensions.

Modern use of Q-learning usually finds the method coupled with deep learning. Similar to the original trajectory of Q-learning. Deep Q-Learning Networks (DQN) have moved from discrete spaces [5] to continuous spaces [6].

Seyde *et al* [7] recognized that the impressive results realized by DQN and its variants might be useful for continuous, multidimensional control problems. Their approach was simple; decompose the the problem and assign one Q-agent to each of the problem's dimensions, effectively reducing the dimensionality of each agent; a decoupled Q-learning Network (DecQN). In this architecture, each agent learns the value function of a single actuator on a robot, but all actuators jointly optimize the given objective. The Q-value is simply the average of all the agents' Q-values for a given state. This is centralized learning. Likewise, the agents all share observations. Centralized learning and shared observations are enough on their own to induce coordination between agents, without additional communication architecture between agents. Though the states are continuous, the authors do not even attempt to provide continuous output capability, but achieve state of the art results on continuous control problems [8] with three output levels per actuator only. The authors do not state whether the dynamics of the problems provide any filtering or damping for the extremely low resolution control output of their agents.

Kumar *et al* [9] extend DQN by creating a pipeline of neural network architectures. Their pipeline depends on an Impala-CNN [10] to extract features from Atari pixel input. The value function parameters are estimated by a ResNet-style [11] network. These parameters determine the shape of C51 [12] value function representation. The authors normalize the output of the learned embedded features in order to achieve positive transfer of performance across Atari games. After transfer, the network architecture quickly adapts to the specific dynamics of a previously unseen game.

Many Q-learning methods update individual Q-values, or relevant groups of values [13], directly from experience. Alternatively, the gradient of the Q-values can be computed with respect to some appropriate objective, and the entire Q-function can be updated by gradient ascent [14], [15].

Kostrikov *et al* [16], rather than optimizing action value according to the Q-function as part of the update step, instead define a temporal difference loss between some expectile value of a state and the value of policy (s, a). The authors call this method Implicit Q-Learning because it approximates policy improvement, but only using (s, a) found in the dataset. (The usual implementation of Q-learning attempts to maximize value according to the current Q-function, which may lead to out-of-distribution actions.) The Q-function is updated by descending the gradient of this novel loss. Large improvement over the dataset performance is demonstrated on 2D navigation and locomotion tasks despite remaining within the dataset distribution of (s, a).

Gradient ascent is also applicable to updating a Q-table that represents the parameters of a Q-function. This update is known as fitted value iteration, because it attempts to fit the optimal Q^* with an approximator, or a combination thereof, rather than approach it in discrete update steps. Li *et al* [17] provide a proof of convergence for fitted value operation for continuous problems, and increase its efficiency by only considering actions that are admissible at any given state.

Lutter *et al* [18] developed a method of fitted value iteration that is agnostic to the timestep length of the simulation or sensor update. The key insight from their work is the inclusion of the control matrix $B(\mathbf{x}, \theta)$ in the expression of the value function. The gradient of the parameters θ optimizes value while accounting for the state transition at the same time, preventing gradient ascent from seeking parameter values that do not make physical sense. The Q-function is represented by a neural network, making this a DQN variant.

Policy-gradient methods are similar to fitted value iteration in that the parameters of a representative function are optimized by moving along the gradient of a performance metric with respect to the parameters. They skip value iteration entirely by representing the policy directly as the action probabilities of a policy $\pi_\theta(a \mid s)$ given the present state. Degris *et al* [19] extend previous policy-gradient methods (which can have quadratic complexity [20]) by creating a more efficient variant with eligibility traces and a time-complexity that does not increase over time, unlike previous methods [21].

Proximal Policy Optimization [22] (PPO) is another policy-gradient method. The authors of PPO intended it to overcome shortcomings in Q-Learning. Petrazzini and Antonelo [23] still optimize the policy directly, but define the PPO objective as a function of the Q-Learning temporal difference error. The authors make a further departure from PPO by parameterizing a combination of beta distributions [24] as the basis function rather than Gaussians. Beta distributions have a finite domain. This gives them an advantage over Gaussian distributions when representing the policy of a physical system. Gaussian distributions have an infinite domain, and will therefore assign nonzero probabilities to actions outside of the physical limits of actuators.

The methods studied in this work rely on regression kernels over a static, regular grid discretization of the (s, a) space. These is not the only options by far. Hiyoshi and Sugihara [25] propose a grid-free interpolation method that is not only continuous, but can be used to generate interpolants with an arbitrary order of continuity, based on Voronoi cells. More recently, Sinclair *et al* [26] offer a way to adaptively partition (s, a) that is based on the density of the samples. A cubic partition is split when the confidence in that partition's estimate is tighter than the metric diameter of the the partition. The average reward and the transition statistics for partitions is maintained in a reasoned way during a split operation, to aid in the calculation of transition probabilities and regret.

II. Methods

A. Training Procedure

We made important changes to the methods and their training in order to obtain adequate and stable performance on the given problem. These changes were necessary, but they do not change the overall structure of the original methods.

We implemented Double Q-Learning [27] for both methods as a means to increase learning stability. This simply means that for every operation in the algorithms that stores a new Q-value ($Q(s, a) \leftarrow Q_{\text{new}}(s, a)$) we instead store it in an alternate lookup table ($Q_{\text{swap}}(s, a) \leftarrow Q_{\text{new}}(s, a)$), then swap all table values for Q and Q_{swap} after a period of training. This is more stable than online learning because one "agent" is acting on *unchanging* offline values while a standby agent learns online. We do not consider this enhancement to change the fundamental structure of the methods, as the means of both querying and modifying table values remains unchanged. Only the table locations have changed. Therefore, comparisons of each method's ability to represent a value function are still valid. Neither work mentions double Q-learning, and it is neither known if it was assumed by the authors that this would be done, nor if their results might have otherwise been improved with its addition.

We further augment each method by choosing actions using the ε-greedy technique [28]. During training this encourages exploration, and during evaluation this prevents the agent from getting stuck in local minima. Again, the underlying structure of Q-table storage and retrieval remains unchanged.

Training occurs in episodes that are grouped into epochs, as in deep learning [29]. Each episode consists of 40 seconds of simulated time separated into discrete timesteps of 0.01 second, for a total of 4000 timesteps per episode. Across each epoch, ε is reduced from 0.50 to 0.05 in equal steps each episode. ε values were determined by manual tuning. Q and Q_{swap} are exchanged after every episode of 4000 timesteps. (Swapping after either 2 or 5 episodes resulted in worse performance.) Each agent was trained for 64 episodes per epoch.

During training, the top performance across any single episode was tracked. Performance is evaluated as the total time the agent keeps the pole within $0.08\bar{3}\pi$ (15°) of vertical ($\theta = 0.0$). When a greater duration than the previous best is achieved, the Q-table is copied to Q_{best}. This is equivalent to early stopping in deep learning [29]. Very often, Q_{best} outperformed the Q-table that underwent the full training regimen.

Both methods used Equation 1 as the reward function during training. The first term rewards the agent for having the pole vertically up and applies a penalty when the pole is vertically down. The second term rewards large angular velocity when the pendulum is down vertically and penalizes it when it is up vertically, such that the system learns necessary "swing-up" actions as quickly as possible. The use of cosine ensures the reward function is continuous and smooth between peak positive and peak negative rewards. The coefficients were determined by manual tuning.

$$R_t = 100\cos(\theta) - 8|\dot{\theta}|\cos(\theta) - R_x \tag{1}$$

$$R_x = \begin{cases} x, & \text{if } x > x_{\text{max}} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

$$x_{\text{max}} = 2.0$$

Even after tuning the first two terms of R_t, performance was lackluster. Visualization of episodes revealed that the agent would swing up, then as soon as the pole began to fall in one direction, the agent would rush off in that direction in an attempt to get the cart under the pole. We subtracted the penalty R_x from the reward to prevent the cart from "wandering". The penalty prevented significant x drift while increasing performance at the same time.

B. Interpolation

Both the methods studied in this work similarly represent the Q-function as an interpolation between entries in the traditional discretized Q-table. This structure still imposes the responsibility of choosing the appropriate grid size on the practitioner. However, the use of interpolation frees the practitioner from having to choose a grid that captures every relevant state and action. Instead, the aim of discretization is having enough points to reconstruct the shape of the optimal Q-function via the chosen regression method.

Approximation by interpolation does come at a cost; first in retrieving the input values for interpolation, then in calculating the desired output value. In discrete Q-learning, each (s, a) state becomes the "address" of the stored Q-value, making value retrieval trivial. In interpolated Q-learning, each value query is associated with at least one nearest-neighbor search amongst all grid points. We accelerate this search by using the k-D [30] and ball [31] spatial trees publicly available in the `NearestNeighbors.jl` Julia package. The number of interpolation points used to reconstruct a Q-value is also a speed-accuracy trade-off that must be made to suit the application. The interpolation calculations in the studied methods are straightforward. However, there are methods with heavier [4] interpolation calculation costs that must be paid for each value query.

C. Method 1: "Q-learning with nearest neighbors." (NNQL), Shah and Xie, 2018 [13]

Shah and Xie propose a number of interpolation techniques in their work [13] and prove convergence guarantees. In order to make these guarantees, Shah and Xie require that interpolation be performed by a *regression kernel*. The kernel must satisfy Equations 3 and 4 in order to be considered a regression kernel in this context.

$$\kappa(x, y) = 0 \quad \text{if} \quad \|x - y\| \geq h \tag{3}$$

$$\sum_{i=1}^{n} \kappa(x, c_i) = 1 \tag{4}$$

In Equations 3 and 4, x is the state for which the value is being estimated and c_i are input Q-table entries chosen by the kernel. The approximated value of Q-state is determined by the Non-parametric Regression Operator, Equation 5. It is simply a weighted average of Q-table entries, with weights given by the selected kernel.

$$\Gamma_{NN,Q}(x, a) = \sum_{i=1}^{n} [\kappa(x, c_i) Q(c_i, a)] \tag{5}$$

The Joint Bellman Operator (Equation 6) describes the learning mechanism. The new value at grid point (c_i, a) is the reward for the action at that state plus the discounted expectation of the kernel-estimated value of the optimal action at the next state.

$$G_Q(c_i, a) = r(c_i, a) + \gamma \mathbb{E}\left[\max_{b \in A} [\Gamma_{NN,Q}(x', b) \mid c_i, a]\right] \tag{6}$$

However, the authors forego keeping track of $\mathbb{E}[\max \Gamma]$, opting instead to update the Q-table entry with an exponential average filter, Equation 7. The fraction of new information α_k accepted into the entry depends on how many grid values N_k are being updated and discount value γ.

$$q^{k+1}(c_i, a) = (1 - \alpha_k) q^k(c_i, a) + \alpha_k G_Q(c_i, a) \tag{7}$$

$$\alpha_k = \frac{\beta}{\beta + k} \tag{8}$$

$$\beta = \frac{1}{1 + \gamma} \tag{9}$$

Policy 1 on page 7 of [13] describes the full details of Nearest Neighbor Q-Learning (NNQL), but we will sketch the main points here. First the agent selects an action a_t according to policy based on the current state $\mathbf{x}_t$. Then, the neighborhood of grid points $c_i \in \mathcal{B}_i$ around $\mathbf{x}_t$ is selected according to the regression kernel being used. A weighted average G_Q of contributions from $q^k(c_i, a)$ is computed according to the regression kernel. Then the value update G_Q is blended back into the members of $\mathcal{B}_i$ with an exponential averaging filter, Equation 7.

D. Nearest Neighbor Regression Kernels

Shah and Xie give three examples regression kernels that satisfy Equations 3 and 4.

a) k-Nearest Neighbor (k-NN) Regression: k-NN Regression assigns equal weight to the values at the k nearest grid values. k-NN queries may be made with a kD-tree [30].

$$\kappa(x, c_i) = \frac{1}{k} \mathbb{1}\{\rho(x, c_i) \leq \rho(x, c_k(x))\} \tag{10}$$

b) Fixed-Radius Near Neighbor Regression: Fixed-Radius Near Neighbor Regression assigns equal weight to every grid value within radius h. Radius NN queries may be made with a ball tree [31].

$$\kappa(x, c_i) = \frac{\mathbb{1}\{\rho(x, c_i) \leq h\}}{\sum_{j=1}^{n} \mathbb{1}\{\rho(x, c_i) \leq h\}} \tag{11}$$

c) Truncated Gaussian Kernel Regression: Truncated Gaussian Kernel Regression takes into consideration only grid points within $d \cdot h$ of x, hence "truncated". Here h is a scaling factor relevant to the problem (using the grid spacing h_g is convenient, thus $h = h_g$), and d is a factor that determines how many h-lengths will be considered when determining neighbors of $(\mathbf{x}, a)$. Values at grid points within that radius account for a fraction of the total estimated value that falls exponentially with distance. We use a ball tree [31] so that only points within a hypersphere of radius $d \cdot h$ are included in each value calculation.

$$\kappa(x, c_i) = \frac{\phi\left(\frac{\rho(x,c_i)}{h}\right)}{\sum_{j=1}^{n} \phi\left(\frac{\rho(x,c_i)}{h}\right)} \tag{12}$$

$$\phi(s) = \exp\left(-\frac{s^2}{2}\right) \mathbb{1}\{s \leq d\} \tag{13}$$

$$d = 1.0$$

E. Method 2: "Continuous-state reinforcement learning with fuzzy approximation", Buşoniu, Lucian, et al., 2008 [3]

Buşoniu *et al* present a fuzzy regression kernel based on the Manhattan distance between each grid point and the point for which the value is being estimated. Like [13], the kernel weighting $\mu_i(x)$ across N grid value inputs x_i (called "cores" by [3]) must sum to 1.

$$\sum_{i=1}^{N} [\mu_i(x)] = 1, \forall x \in X \tag{14}$$

$$\exists x_i \text{ for which } \mu_i(x_i) = 1 \tag{15}$$

The authors treat each core x_i as a membership class. When $x = x_i$, x is fully a member of x_i and of no other class. When $x \neq x_i$, x may be a member of several classes. The degree of membership of each class determines the weight assigned to the value at the corresponding core when estimating the value of x. Unfortunately, the exact equation for $\mu_i(x)$ is not given. The reader is only told that it is a normal fuzzy set partition (the highest membership value / weight is 1) and given the rules defined by Equations 14 and 15. The normalized inverse of the Manhattan distance from x_i (Equation 17) would yield the triangular fuzzy partition (Equation 16) depicted in Figure 1 on page 12 of [3]. Limiting the spatial tree search to $k \leq 2N$ nearest neighbors in an N-dimensional search ensures that only neighbors within Nh of the query point will be returned. In our work we found $k = 3$ to yield the best results, though we tested others (See Figure 1).

$$\kappa(\mathbf{x}, \mathbf{x}_i) = \frac{\left(\|\mathbf{x} - \mathbf{x}_i\|_M\right)^{-1} \mathbb{1}\{\|\mathbf{x} - \mathbf{x}_i\|_M \leq Nh\}}{\sum_{j=1}^{n} \left(\|\mathbf{x} - \mathbf{x}_i\|_M\right)^{-1} \mathbb{1}\{\|\mathbf{x} - \mathbf{x}_i\|_M \leq Nh\}} \tag{16}$$

$$\|\mathbf{y}\|_M = \sum_{k=1}^{N} |y_k| \tag{17}$$

Buşoniu *et al* use the entries of the Q-table as the parameters $\theta_{i,j}$ of a linear basis function representing the value function (Equation 18). Instead, we represent the value of a state-action pair directly as $Q(\mathbf{x}, a)$ (Equation 19). Equation 18 operates on an array of θ_i linear basis functions, while Equation 19 operates on an array of scalar values.

$$\theta_{i,j} \leftarrow \rho(x_i, u_j) + \gamma \max_j \left[\sum_{i=1}^{N} \left[\mu_i\left(f\left(x_i, u_j\right)\right) \theta_{i,j}\right]\right] \tag{18}$$

$$Q(\mathbf{x}_t, a_t) \leftarrow r(\mathbf{x}_t, a_t) + \gamma \max_a \left[\sum_{i=1}^{N} \left[\mu_i\left(f\left(\mathbf{x}_t, a_t\right)\right)\right]\right] \tag{19}$$

Note that in Equation 19, only the grid point nearest $(\mathbf{x}_t, a_t)$ is updated, and not a neighborhood of grid points as in [13]. We thought this might be an error, so we tested a number of methods of distributing the value update back into the grid points of the neighborhood within Nh of $(\mathbf{x}_t, a_t)$. None of these worked as well as the update via Equation 19, adapted from the authors' work.

III. Experiment

The experimental setting is the classical Cartpole environment, but as defined by Florian [32] instead of Barto *et al* [33]. Florian [32] gives the governing Equations 23 to 25 for the Cartpole environment. The authors claim their state equations model the physical Cartpole system more correctly, but verifying that claim is not the purpose of this work. The state is defined by the column vector $\mathbf{x}$ The dynamics are entirely deterministic and depend only on the previous state $\mathbf{x}_{t-1}$ and the current input F_t.

$$\mathbf{x} = \begin{bmatrix} \ddot{\theta} \\ \dot{\theta} \\ \theta \\ \ddot{x} \\ \dot{x} \\ x \\ N_c \end{bmatrix} \tag{20}$$

However, for learning purposes a reduced form of $\mathbf{x}$ is used (Equation 21), made up of the most relevant elements. Note that while state variable x was used in Equation 2 to penalize drift, it was not required to learn control policies. We determined the necessary state variable ranges by applying sine waves of input forces with a magnitude of 10N and frequencies of {16.0, 14.25, 12.8, 11.6, 10.6, 9.1, 8.0, 5.3, 4.0, 2.0, 1.0, 0.5, 0.25, 0.125}Hz, each for 40 seconds (4000 timesteps), and observed the maximum absolute values of each state variable in Equation 21.

$$Q(\mathbf{x}, a) = \begin{bmatrix} \ddot{\theta} \in [-30.0, +30.0] \\ \dot{\theta} \in [-15.0, +15.0] \\ \theta \in [-20.0, +20.0] \\ \dot{x} \in [-10.0, +10.0] \\ \hline F \in [-10.0, +10.0] \end{bmatrix} \tag{21}$$

In the governing Equations 23 to 25, the following parameters were used:

$$g = 9.81 \qquad \ell = 0.5$$
$$m_c = 1.0 \qquad \mu_c = 1.25(10^{-4})$$
$$m_p = 0.1 \qquad \mu_p = 5.00(10^{-7})$$

Integrated state variables $\{\dot{\theta}, \theta, \dot{x}, x\}$ were obtained simply by applying the trapezoidal numeric integration rule, Equations 26 to 29. Even though this integration method violates conservation of both energy and angular momentum, the system exhibits smooth and believable behavior at the chosen timestep of 0.01 seconds. (e.g. No grossly overdamped or runaway effects were observed.)

After training as described in Section II-A has completed, then Q_{best} is evaluated at $\varepsilon = \{0.00, 0.01, 0.025, 0.05, 0.0625, 0.075, 0.0875, 0.10, 0.125, 0.1375, 0.15, 0.1625, 0.175, 0.1875, 0.20\}$ for 100 episodes at each ε. The best average score out of the tested 'ε's is reported in Figures 1, 2, and 3. Note that the ε at which the best average score occurred is not reported, though this was typically below 0.15.

IV. Results

A. Method 1: "Q-learning with nearest neighbors." (NNQL), Shah and Xie, 2018 [13]

In Figures 1 and 2 the following hyperparameter were used except where otherwise specified. h_g refers to grid spacing and h refers to the Gaussian scaling parameter given by Equation 12.

$$\gamma = 0.95 \qquad h_g = 4.0 \qquad k = 3$$
$$\text{Epochs} = 64 \qquad \text{Episodes} = 64$$

Param.	Run 1	Run 2	Run 3	Max.	Avg.
$\gamma = 0.99$	0.269	0.250	0.297	0.297	0.272
$\gamma = 0.95$	1.521	**19.560**	0.252	19.560	7.111
$\gamma = 0.75$	0.035	0.102	0.181	0.181	0.106
$h_g = 7.5$	0.759	0.365	0.172	0.759	0.432
$h_g = 2.0$*	0.264	0.962	0.781	0.962	0.669

Fig. 1. Results of hyperparameter search for [13], kNN regression kernel, *Trained for 128 epochs

Param.	Run 1	Run 2	Run 3	Max.	Avg.
$k = 2$*	0.220	**3.130**	0.659	3.130	1.336
$k = 4$*	0.263	0.285	0.192	0.285	0.247
$k = 5$*	0.298	0.070	0.206	0.298	0.191
$r = 4.5$ †	0.112	0.060	0.015	0.112	0.062
$r = 7.0$ †	0.218	0.074	0.074	0.218	0.122
$r = 10.0$ †	0.288	0.276	0.286	0.288	0.283
$h = 4.5$ ‡	0.081	0.098	0.373	0.373	0.184
$h = 7.0$ ‡	0.163	0.224	0.302	0.302	0.230

Fig. 2. Results of hyperparameter search for [13], * kNN kernel, †fixed radius kernel, ‡truncated Gaussian regression kernel

B. Method 2: "Continuous-state reinforcement learning with fuzzy approximation", Buşoniu, Lucian, et al., 2008 [3]

In Figure 3 the following hyperparameter were used except where otherwise specified.

$$\gamma = 0.99 \qquad h_g = 4.0 \qquad k = 3$$
$$\text{Epochs} = 64 \qquad \text{Episodes} = 64$$

Param.	Run 1	Run 2	Run 3	Max.	Avg.
$\gamma = 0.99$	3.687	0.324	19.577	**19.577**	7.863
$\gamma = 0.95$	1.308	1.061	4.051	4.051	2.140
$\gamma = 0.75$	0.963	0.707	1.912	1.912	1.194
$h_g = 7.5$	0.578	1.840	0.280	1.840	0.899
$h_g = 2.0$*	0.529	0.147	0.690	0.690	0.455
$h_g = 1.0$*	0.320	0.173	0.780	0.780	0.424
$k = 2$	0.312	6.362	0.498	6.362	2.391
$k = 4$	0.550	1.488	29.539	**29.539**	10.526
$k = 5$	0.597	25.690	2.606	25.690	9.631

Fig. 3. Results of hyperparameter search for [3], *Trained for 128 epochs

V. Conclusions

There are two sub-tasks to the Cartpole Problem; swing-up, and balancing. A good controller should contain sufficient experience and richness to encode and perform both sub-tasks. All tested controllers were able to achieve swing-up, and all tested hyperparameters resulted in agents that were able to bring the pole to near-vertical, if only for a few frames. This seems to be the trivial sub-task, as noted by Barto and Sutton [28]. The tested methods and hyperparameter settings were much less successful at the balancing sub-task, with all but a few settings achieving less than a full second of balancing within 15° of vertical ($\theta = 0°$).

Interestingly, the tested controllers that solved the balancing sub-task did so with overwhelming success and double-digit balancing times. There seemed to be no continuum between the successful few and the paltry average. The best-performing agent was a fuzzy controller [3] with $k = 4$, $\gamma = 0.99$, and $h_g = 4.0$, which balanced for nearly $\frac{3}{4}$ of the 40 second episode period. This was surprising to us because we consider 4.0 to be an extremely course grid spacing. Though impressive, that level of performance was very intermittent, and two other runs of an identical training regimen produced agents with one and two orders of magnitude poorer performance.

We believe that after 16.4 million training steps, with significant time devoted to exploration, many more of controllers contained the necessary experience to perform balancing than are shown in the results. Qualitatively, the most common problem we observed was an agent dithering near the downward position ($\theta = \pi$) for long periods without the forceful cart oscillations required to achieve a swift swing-up. Rather, many agents would achieve a slow swing-up by applying only a little force each swing, leaving little time for balancing. Even if the Q-table contained high-value actions to maintain balance, these remained hidden behind a woefully sub-optimal policy accessible at the episode start ($\theta = \pi$). We believe that this problem can be solved by applying Temporal Difference and/or Eligibility Trace tactics from the early development of Q-learning [28] (See Section VI-B). It is also possible that $\theta = \pi$ is a critical region of the problem that is highly sensitive to small differences in action F, in which case a non-uniform discretization approach, as in [26], is warranted. We are unsure why the authors of [3], [13] did not face similar barriers.

VI. Extensions

A. Failed Attempts

a) Curriculum Learning: Inspired by curriculum learning methods in robotics [34] and visual question answering [35], we ran the fuzzy method [3] on a curriculum of sine waves of input forces with a magnitude of 10N and frequencies of {16.0, 14.25, 12.8, 11.6, 10.6, 9.1, 8.0, 5.3, 4.0, 2.0, 1.0, 0.5, 0.25, 0.125}Hz, each for 40 seconds (4000 timesteps). During curriculum learning, no regard was given to rewards other than to update the Q-table as described in [3]. This procedure resulted in worse performance of the following ε-greedy training and was not pursued further, even though one of the example episodes includes a successful swing-up.

b) Composite Q-Table: Noting that in both methods the Q_{best} table was most often less populated than either Q or Q_{swap}, we wished to recover training information that was gained after caching Q_{best}. (See Section II-A for definition of Q_{best} and the evaluation metric used.) To this end, we also tracked the best average score across epochs, and saved Q as Q_{BAv} at the end of an epoch whenever the previous best average score was exceeded. At evaluation time, we scored Q_{best} and Q_{BAv} at $\varepsilon = 0.035$ and designated the best-performing of the two as the "primary" table. We then constructed a Q-table with the values of the primary table, and with any zero entries of the primary filled in with corresponding non-zero entries of the other table; forming an augmented table Q_{Aug}. We then compared the performance of Q and Q_{swap} at the end of a training regimen and designated the best of the two as the "secondary" table. Corresponding non-zero entries of the secondary table were used to fill in remaining zero entries of Q_{Aug}, followed by the other table. The final Q_{Aug} consistently performed better than either Q or Q_{swap}, and intermittently performed an order of magnitude better than Q_{best}. However, in most instances Q_{Aug} and Q_{best} performed similarly, with Q_{Aug} lagging slightly in performance. Q_{Aug} performance did not exceed Q_{best} often enough to be of interest. Rather, other, more-established methods of bootstrapping should be pursued.

B. Easy Extensions

The following can easily be applied to agents with the structure described in Sections II-C and II-E without impacting their ability to solve control problems in continuous (s, a) spaces. They may even help resolve the difficulties described in Section V. However, development difficulties (Section VIII) restricted our ability to implement them.

a) Off-Policy Temporal Difference, One-Step: The Temporal Difference learning rule is given by Equation 22.

$$Q'(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a_{\text{opt}}} \left[Q(s_{t+1}, a_{\text{opt}}) \right] - Q(s_t, a_t) \right) \tag{22}$$

It may be used in place of Equations 6 and 19 from [13] and [3], respectively, provided that $Q(s_t, a_t)$ terms on the right-hand side are obtained by the appropriate interpolation, and the final $Q'(s_t, a_t)$ value from the left-hand side is stored according to the associated method.

b) Bootstrapping, Q(λ) [36]: We can back up traces from the greedy portions of learning episodes. That is, the (s, a) pairs in the Q-table leading up to a high-value state are increased in value. We can also do this by running full episodes with $\varepsilon = 0.0$, finding the peaks in value (Equation 1), and backing up these segments.

VII. Future Work

a) Adapt a semi-continuous DQN to be fully continuous: It would be interesting to adapt a method with continuous input, but discrete output, such as [7], to have fully continuous output, using the best interpolation hyperparameters from [13] or [3].

b) Adapt either [13] or [3] to a different function representation: Likewise, it would be interesting to update the traditional Q-function representation of [13] and [3] to one suited to more complex problems, such as [16].

Governing Equations for the Cartpole Environment [32]

$$N_c = (m_c + m_p)g - m_p\ell(\ddot{\theta}\sin(\theta) + \dot{\theta}^2\cos(\theta)) \tag{23}$$

$$\ddot{\theta} = \frac{g\sin(\theta) + \cos(\theta)\left(\frac{-F - m_p\ell\dot{\theta}^2(\sin(\theta) + \mu_c\,\mathrm{sgn}(N_c\dot{x})\cos(\theta))}{m_c + m_p} + \mu_c g\,\mathrm{sgn}(N_c\dot{x}) - \frac{\mu_p\dot{\theta}}{m_p\ell}\right)}{\ell\left(\frac{4}{3} - \frac{m_p\cos(\theta)}{m_c + m_p}\left(\cos(\theta) - \mu_c\,\mathrm{sgn}(N_c\dot{x})\right)\right)} \tag{24}$$

$$\ddot{x} = \frac{F + m_p\ell\left(\dot{\theta}^2\sin(\theta) - \ddot{\theta}\cos(\theta)\right) - \mu_c N_c\,\mathrm{sgn}(N_c\dot{x})}{m_c + m_p} \tag{25}$$

$$\dot{\theta}_t \leftarrow \dot{\theta}_{t-1} + \frac{\ddot{\theta}_{t-1} + \ddot{\theta}_t}{2}\Delta t \tag{26}$$

$$\theta_t \leftarrow \theta_{t-1} + \frac{\dot{\theta}_{t-1} + \dot{\theta}_t}{2}\Delta t \tag{27}$$

$$\dot{x}_t \leftarrow \dot{x}_{t-1} + \frac{\ddot{x}_{t-1} + \ddot{x}_t}{2}\Delta t \tag{28}$$

$$x_t \leftarrow x_{t-1} + \frac{\dot{x}_{t-1} + \dot{x}_t}{2}\Delta t \tag{29}$$

$$\Delta t = 0.01 \text{ seconds}$$

VIII. Project Postmortem

In this section we outline some major issues that sapped the potential of the project and ultimately provided no intellectual value.

a) Project change: A partial implementation of metric learning method [4] was built and then abandoned in favor of the two chosen methods. Also, two unused training environments were created.

b) Language change: The first iteration of the project was built using Python due to familiarity with the language. The entire project was then switched to Julia to take advantage of its speed, to access related projects written in Julia, and to support a potential collaboration that did not bear fruit. A comparison of the dis/advantages of each language before development began could have prevented this loss of time. In fact, the latter two reasons turned out not to be relevant.

c) Undue focus on tooling: Several restarts of the project stemmed from building up data and code structures before they were strictly needed. The anticipated need for such was overestimated. We later found that a "flat and fast" approach, one that eschews over-engineering and instead embraces the YAGNI (You Ain't Gonna Need It) principle, is most suited for exploratory development. Indeed, putting too much effort into code re-use and generalization (the DRY (Don't Repeat Yourself) principle) results in the rapid accumulation of "technical debt" [37] by basically constructing an API [38] for algorithms that do not yet exist.

Examples of over-engineering we accumulated, but did not use:

- A custom implementation of a priority queue in Python.
- A second order function that produces linear Newton interpolant functions from an arbitrary set of $\langle x, y\rangle$ pairs. This included learning about Julia metaprogramming, with which we were unfamiliar.
- A wrapper around `KDTree` from `scipy.spatial` that allowed points to be added dynamically. This implementation had a collection of "loose" points that would be searched in a linear fashion and compared to the `KDTree` results. When the collection reached size N, the spatial tree would be rebuilt with all points and the side collection would be emptied. The hope was to amortize the cost of building the tree by coupling an inefficient search over a small collection with an efficient search over a large collection. We built this in service of a dynamic Q-function partitioning method that never came to fruition.
- An attempt to modularize the use of different non-parametric regression kernels using the Strategy Design Pattern [39]. This structure was much larger than the few lines of code that needed to be changed in order to actually switch kernels.
- A deeply nested class hierarchy was built in Python, then abandoned.
- A deeply nested class hierarchy was built in Julia, then abandoned. This included learning OOP for Julia.⧫ In fact, it is now our opinion that classes, and the entire Object Oriented framework, should be avoided in all cases, except when; abstraction adds necessary clarity over unstructured code, or it is absolutely certain that structures will be re-used many times in a way that clearly and concisely supports solving the problem at hand.

Phase 2: Extension of Two Continuous Q-Learning Methods

Project Goals

- Improve the disappointing performance of [13] and [3]
  - Temporal Difference Learning Rule
  - $Q(\lambda)$ Eligibility Traces
- Implement one of each of the following:
  - Fitted Value Iteration
  - Policy Gradient
  - Metric Learning
  - DQN
  - Continuous DQN
- The CORVID Project
  - Model-based Reinforcement Learning with an explicit multi-dimensional model

IX. Continuous Q-Learning Performance Improvement

A. *Temporal Difference*

B. *$Q(\lambda)$ Eligibility Traces*

References

[1] C. J. C. H. Watkins, "Learning from delayed rewards," 1989.

[2] T. Horiuchi, A. Fujino, O. Katai, and T. Sawaragi, "Fuzzy interpolation-based q-learning with continuous states and actions," in *Proceedings of IEEE 5th International Fuzzy Systems*, vol. 1. IEEE, 1996, pp. 594–600.

[3] L. Buşoniu, D. Ernst, B. De Schutter, and R. Babuška, "Continuous-State Reinforcement Learning with Fuzzy Approximation," in *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*, K. Tuyls, A. Nowe, Z. Guessoum, and D. Kudenko, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, vol. 4865, pp. 27–43, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-540-77949-0_3

[4] M. S. Emigh, E. G. Kriminger, A. J. Brockmeier, J. C. Príncipe, and P. M. Pardalos, "Reinforcement learning in video games using nearest neighbor interpolation and metric learning," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 1, pp. 56–66, 2014.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[6] A. Srinivas, M. Laskin, and P. Abbeel, "Curl: Contrastive unsupervised representations for reinforcement learning," *arXiv preprint arXiv:2004.04136*, 2020.

[7] T. Seyde, P. Werner, W. Schwarting, I. Gilitschenski, M. Riedmiller, D. Rus, and M. Wulfmeier, "Solving continuous control via q-learning," *arXiv preprint arXiv:2210.12566*, 2022.

[8] S. Tunyasuvunakool, A. Muldal, Y. Doron, S. Liu, S. Bohez, J. Merel, T. Erez, T. Lillicrap, N. Heess, and Y. Tassa, "dm_control: Software and tasks for continuous control," *Software Impacts*, vol. 6, p. 100022, 2020.

[9] A. Kumar, R. Agarwal, X. Geng, G. Tucker, and S. Levine, "Offline q-learning on diverse multi-task data both scales and generalizes," *arXiv preprint arXiv:2211.15144*, 2022.

[10] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International conference on machine learning*. PMLR, 2018, pp. 1407–1416.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[12] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," in *International Conference on Machine Learning*. PMLR, 2017, pp. 449–458.

[13] D. Shah and Q. Xie, "Q-learning with Nearest Neighbors," *arXiv:1802.03900 [cs, math, stat]*, Oct. 2018, arXiv: 1802.03900. [Online]. Available: http://arxiv.org/abs/1802.03900

[14] C. K. Tham and R. W. Prager, "A modular q-learning architecture for manipulator task decomposition," in *Machine Learning Proceedings 1994*. Elsevier, 1994, pp. 309–317.

[15] A. G. Barto, "Chapter 2 - reinforcement learning," in *Neural Systems for Control*, O. Omidvar and D. L. Elliott, Eds. San Diego: Academic Press, 1997, pp. 7–30. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780125264303500039

[16] I. Kostrikov, A. Nair, and S. Levine, "Offline reinforcement learning with implicit q-learning," *arXiv preprint arXiv:2110.06169*, 2021.

[17] H. Li, S. Shao, and A. Gupta, "Fitted value iteration in continuous mdps with state dependent action sets," *IEEE Control Systems Letters*, vol. 6, pp. 1310–1315, 2021.

[18] M. Lutter, S. Mannor, J. Peters, D. Fox, and A. Garg, "Value iteration in continuous actions, states and time," *arXiv preprint arXiv:2105.04682*, 2021.

[19] T. Degris, P. M. Pilarski, and R. S. Sutton, "Model-free reinforcement learning with continuous action in practice," in *2012 American Control Conference (ACC)*. IEEE, 2012, pp. 2177–2182.

[20] J. Peters and S. Schaal, "Natural actor-critic," *Neurocomputing*, vol. 71, no. 7-9, pp. 1180–1190, 2008.

[21] R. S. Sutton, S. D. Whitehead *et al.*, "Online learning with random representations," in *Proceedings of the Tenth International Conference on Machine Learning*, 2014, pp. 314–321.

[22] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[23] I. G. Petrazzini and E. A. Antonelo, "Proximal policy optimization with continuous bounded action space via the beta distribution," in *2021 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 2021, pp. 1–8.

[24] M. Abramowitz and I. A. Stegun, *Handbook of mathematical functions with formulas, graphs, and mathematical tables*. US Government printing office, 1964, vol. 55.

[25] H. Hiyoshi and K. Sugihara, "Voronoi-based interpolation with higher continuity," in *Proceedings of the sixteenth annual symposium on Computational geometry*, 2000, pp. 242–250.

[26] S. Sinclair, T. Wang, G. Jain, S. Banerjee, and C. Yu, "Adaptive discretization for model-based reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 3858–3871, 2020.

[27] H. Hasselt, "Double q-learning," *Advances in neural information processing systems*, vol. 23, 2010.

[28] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction," *Robotica*, vol. 17, no. 2, pp. 229–235, 1999.

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[30] J. L. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun. ACM*, vol. 18, pp. 509–517, 1975.

[31] S. M. Omohundro, *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[32] R. V. Florian, “Correct equations for the dynamics of the cart-pole system,” *Center for Cognitive and Neural Studies (Coneural), Romania*, 2007.

[33] A. G. Barto, R. S. Sutton, and C. W. Anderson, “Neuronlike adaptive elements that can solve difficult learning control problems,” *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 834–846, 1983.

[34] R. Li, M. Roberts, M. Fine-Morris, and D. Nau, “Teaching an htn learner,” *HPlan 2022*, p. 68.

[35] V. Damodaran, S. Chakravarthy, A. Kumar, A. Umapathy, T. Mitamura, Y. Nakashima, N. Garcia, and C. Chu, “Understanding the role of scene graphs in visual question answering,” *arXiv preprint arXiv:2101.05479*, 2021.

[36] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[37] E. Allman, “Managing technical debt,” *Communications of the ACM*, vol. 55, no. 5, pp. 50–55, 2012.

[38] L.-E. Janlert, “Modeling change: The frame problem,” 1987.

[39] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head First Design Patterns: A Brain-Friendly Guide*. ” O’Reilly Media, Inc.”, 2004.