Q-Learning in Continuous State Action Spaces

Alex Irpan
alexirpan@berkeley.edu

December 5, 2015

Contents

1 Introduction

Recent results combining Q-learning with deep neural nets have achieved surprising result. The highest profile success story comes from Google DeepMind's paper on learning how to play Atari games from raw pixel input [1]. However, their approach applies best to problems with a small discrete action space, which is not ideal for many problems. This presents some potential methods for Q-learning in continuous state action spaces.

2 Background

A Markov Decision Process (MDP) is a tuple (S, A, P, R, γ, H). S is the state space, A is the action space, P is the transition dynamics which represents $Pr(s_{t+1}|s, a)$. R represents the reward when taking action a in state s. For now, we assume R is constant given s, a, and successor state s'. $\gamma \in (0, 1]$ is the discount factor. Commonly, exponential discounting is used, meaning the reward at time step t is multiplied by γ^t. H is the horizon, and describes how

many time steps the agent has to act. This can either be finite or infinite; in the infinite horizon case we require $\gamma < 1$. Collectively, these are called the *dynamics* of the MDP.

As suggested by the name, an MDP is memoryless, and the next state depends only on the current state and action. More formally,

$$s_{t+1} \perp\!\!\!\perp s_1, a_1, \ldots, s_{t-1}, a_{t-1} | s_t, a_t$$

A *policy* π returns the action an agent should take from a given state. The goal is to learn the policy that maximizes expected reward (henceforth called the optimal policy and denoted by π^*.

In many scenarios, the dynamics are only partially known. In these settings, we need to learn estimates of the dynamics, which makes the problem significantly more difficult. *Reinforcement learning* algorithms (often abbreviated as RL) are a broad class of algorithms for solving these problems. At a high level, these algorithms are structured as the following.

1. Initialize parameters of the policy randomly
2. Sample the state action space, observing transitions (s_t, a_t, r_t, s_{t+1}) (reward and next state for taking action s_t in state s_t.)
3. Update parameters according to these samples in a way that increases reward of the new policy. Repeat.

We present some traditional RL algorithms before describing their modifications. For more details, see [2].

3 Q-Learning

In the following, we often refer to *episodes* from a policy. An episode is a sequence $(s_0, a_0, s_1, a_1, \cdots)$ of the states and actions the agent observes. An episode continues until we either run out of time or hit a terminal state.

Let $V^\pi(s)$ be the *value function*, the expected reward when following policy π from state s. The *Q-function* $Q^\pi(s, a)$ is the expected reward for taking action a from s, then following policy π for the rest of the episode. It can be defined as

$$Q^\pi(s, a) = E_{s'}[R(s, a) + \gamma V^\pi(s')] = E_{s'}[R(s, a) + \gamma Q(s', \pi(s'))]$$

Q-learning is a reinforcement learning algorithm that learns the Q-function. To derive the approach used, first consider the optimal value and Q-functions. The optimal value function is $V^*(s) = V^{\pi^*}(s)$, and the optimal Q-function is $Q^*(s, a) = Q^{\pi^*}(s, a)$. Because MDPs are memoryless, we have

$$V^*(s) = \max_a Q^*(s, a)$$

In other words, the optimal action a is the one such that the reward for taking action a and acting optimally afterwards is maximized. Substituting into the

Q-function, this gives

$$Q^*(s,a) = E_{s'}[R(s,a) + \gamma \max_{a'} Q^*(s',a')]$$

From this, we can derive the core operator of Q-learning, the Bellman backup operator.

Going forward, we assume Q-functions are defined by a parameter vector θ, with notation $Q_\theta(s,a)$. On each iteration of Q-learning, we have a parameter estimate $\theta^{(n)}$. From the current estimated $Q(s,a)$, generate a policy π_n. One common choice is ϵ-greedy, defined as

$$\pi_n(s) = \begin{cases} \text{random} & \text{with probability } \epsilon \\ \text{argmax}_a \, Q_{\theta^{(n)}}(s,a) & \text{otherwise} \end{cases}$$

After trying action a from state s, we observe reward R and next state s'. After seeing this sample (s,a,R,s'), we want the Q-function to satisfy

$$Q_\theta(s,a) = R + \gamma \max_{a'} Q_\theta(s',a')$$

Use a quadratic loss function to describe the error.

$$\ell^{(n)}(\theta) = \frac{1}{2}((R + \gamma \max_{a'} Q_{\theta^{(n)}}(s',a')) - Q_\theta(s,a))^2$$

Update θ by

$$\theta^{(n+1)} \leftarrow \min_\theta \ell^{(n)}(\theta^{(n)})$$

Note every iteration has a slightly different loss function. Computing $\max_a Q_\theta(s,a)$ is difficult if θ is a parameter of the loss function, so we instead maximize with respect to known parameter $\theta^{(n)}$, and use a more accurate loss function each iteration.

The Bellman backup can be interpreted as a one step lookahead. $R + \gamma \max_{a'} Q_{\theta^{(n)}}(s,a)$ is the value for taking action a, then acting optimally based on the current estimate of θ. Thus, the loss depends on the sample and the value exactly one timestep in the future.

Q-learning is guaranteed to converge when Q is represented by a $|S| \times |A|$ array. Once parametrized, these theoretical guarantees go away, but unfortunately this is necessary to make problems computationally tractable.

For the approach used in the project, the parameters θ are the parameters of a neural net. The minimum θ can be computed by running gradient descent for enough iterations, but instead we run only one update.

$$\theta^{(n+1)} \leftarrow \theta^{(n)} - \eta \nabla_\theta \ell^{(n)}(\theta^{(n)})$$

where η is a step size. Expanding the gradient gives

$$\theta^{(n+1)} \leftarrow \theta^{(n)} - \eta(R + \gamma \max_{a'} Q_{\theta^{(n)}}(s',a') - Q_\theta(s,a)) \nabla_\theta Q_\theta(s,a)$$

This approach is more common, and can be interpreted as either avoiding overfitting to one sample or adding a penalty term for moving too far from $\theta^{(n)}$. It can also be motivated by noting that samples received depend on the accuracy of θ, so updating θ earlier should give a more accurate loss function (at the cost of more variance.)

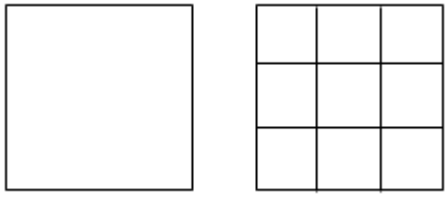
The great strength of reinforcement learning is that it is online and model free. The agent can learn even when it has no information on how the environment behaves, and can do so by directly taking actions in the environment instead of in simultation. This gives these algorithms incredible generality.

However, this generality comes at significant cost. Often, reinforcement learning requires many samples to converge to a good policy. RL also struggles with exploration-exploitation trade-offs, the problem of choosing whether to repeatedly take a known good action or whether to explore an unknown action that could be better.

In recent years, reinforcement learning has received a revival of interest because of the advancements in deep learning. Often called deep reinforcment learning, this approach uses a deep neural net to model Q-functions. The neural net takes state as input, and outputs $|A|$ values, corresponding to the estimated Q-values for each action. Neural net Q-functions still have the same issues as the rest of RL, but more uniquely training a large neural net requires even more samples and even more training time.

4 Q-Learning In Continuous Spaces

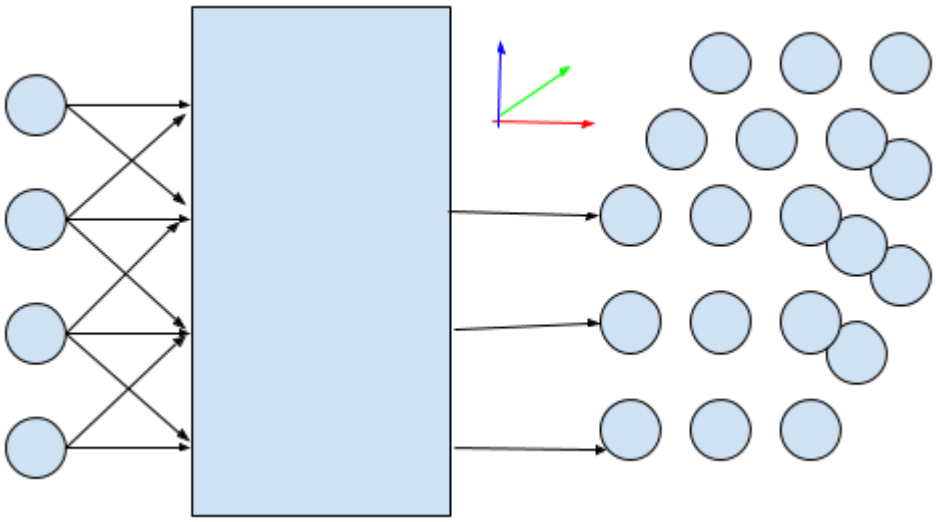
Q-learning in continuous spaces becomes significantly trickier, since many approaches for Q-learning assume a discrete action space. The simplest way to get around this is to apply discretization. For an action from a continuous range, divide it into N buckets. Apply Q-learning to this approximated MDP with a discrete action space. When getting samples, act in the true continuous MDP.



Dividing a two dimensional action space into 9 discrete actions

As N increases, the discretization gets more accurate. However, this introduces new problems. Suppose the MDP action space is multidimensional. If the action space is d-dimensional, and each dimension is discretized into N buckets, there are N^d actions. For even small dimensional problems, this quickly limits how large N can be. For neural net Q-functions, this is especially problematic

because the final layer needs to have N^d outputs, which makes the number of parameters grow exponentially.

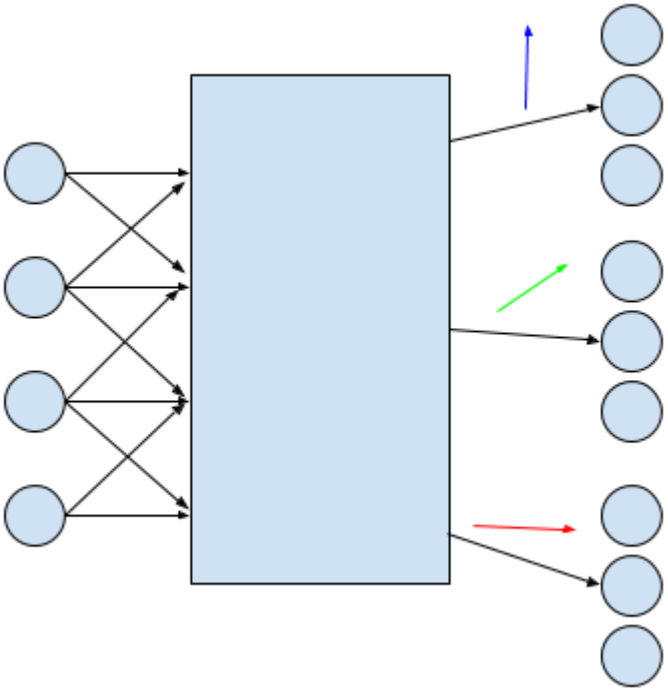


A representation for when $N = 3, d = 3$. The final layer needs 27 outputs.

To deal with the problem, we add indepedence assumptions. More specifically, we assume the dimensions of the action space are mutually independent. Suppose the Q-function is the additive sum of basis functions that are defined only over one of the action dimensions.

$$Q(s, a) = \sum_i Q_i(s, a(i))$$

Then, the value for an action only requires storing the d values for each $Q_i(s, a(i))$, giving Nd outputs total.



The representation with independence assumptions. The 9 outputs are split into three blocks of 3, for the three dimensions of the action space.

To motivate why an additive Q-function is good, consider the Q-learning update. This requires evaluating $Q(s,a), \max_a Q(s,a)$, and $\nabla Q(s,a)$. Computing $Q(s,a)$ is easy from the definition. For the other two,

$$\max_a Q(s,a) = \max_a \sum_i Q_i(s,a(i)) = \sum_i \max_{a(i)} Q_i(s,a(i))$$

$$\nabla Q(s,a) = \nabla \sum_i Q_i(s,a(i)) = \sum_i \nabla Q_i(s,a(i))$$

The additive nature makes everything in the update computable from just the basis functions. Much like directed graphical models, this allows us to implicitly represent the joint action space with a set of smaller spaces, and this representation allows us to run Q-learning without ever needing to explicitly expand out all N^d possible actions.

Note also that the independence assumption arises naturally - the value for the component $a(i)$ can be maximized independently of the value for $a(j)$. Work by Guestrin et. al. showed that the Q-function can be represented by a graph, where edges denote dependence and lack of edges denote conditional independence [3]. Computing the maximum can be done with an analogue of variable elimination.

Adding this independence assumption may not be accurate, and in fact for many problems it seems very unlikely it would be. However, parametrized Q-functions are already quite lossy, so it seemed interesting to see if the increased loss from adding unsupported independences could be balanced out by the faster computation.

5 Experimental Results

For the continuous problem, I have tried running experiments in LQR, because the problem is both small and the dimension can be made arbitrarily large. Unfortunately, I have yet to get Q-learning working in even the one-dimensional case, which makes it hard to run experimenets in higher dimensions. This section will be devoted to analyzing a more discrete problem.

In the Atari domain, the states are the pixels displayed on the screen. The actions are the 18 actions corresponding to a choice of each of the following.

- Left, neutral, or right on the control stick.
- Up, neutral, or down on the control stick.
- Press, do not press the button.

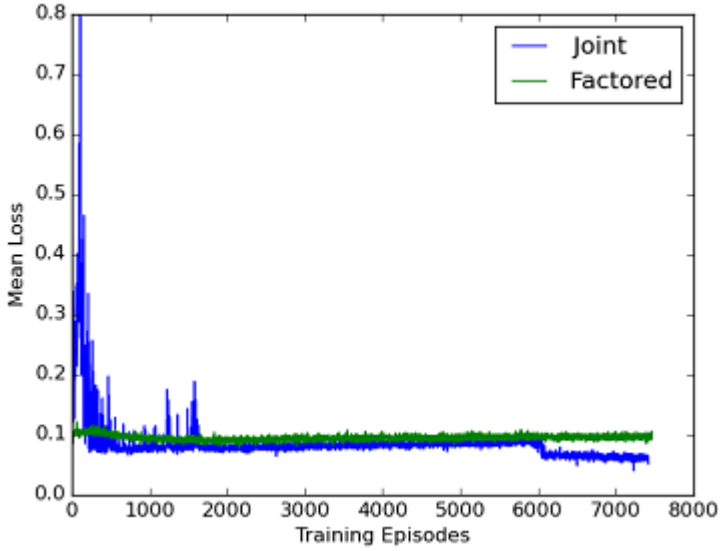
The reward is the score gained after taking the given action.

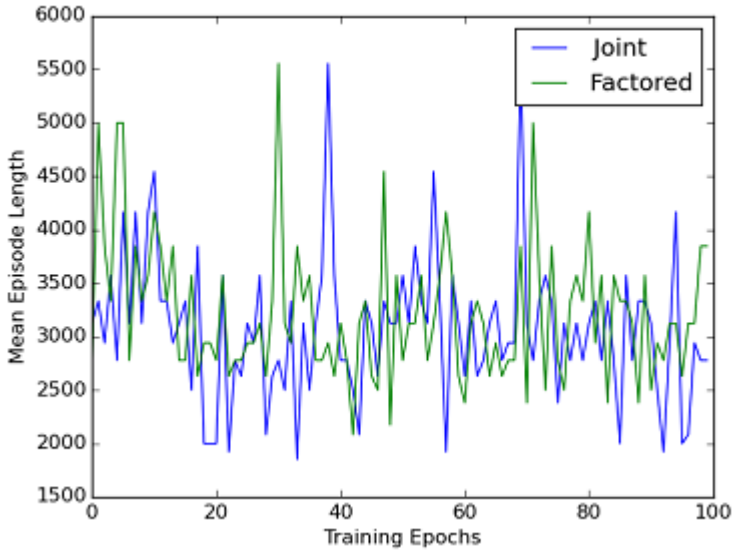
As a control, the experiment starts with the neural net architecture from Google DeepMind. This is a convolutional neural net that ends in a final fully connected layer with 18 outputs, the Q-values for the 18 actions. (They actually use a variant called DQN. This variant stores a replay buffer of the most recent actions, and samples data from the replay buffer instead of using just the most recent action.)

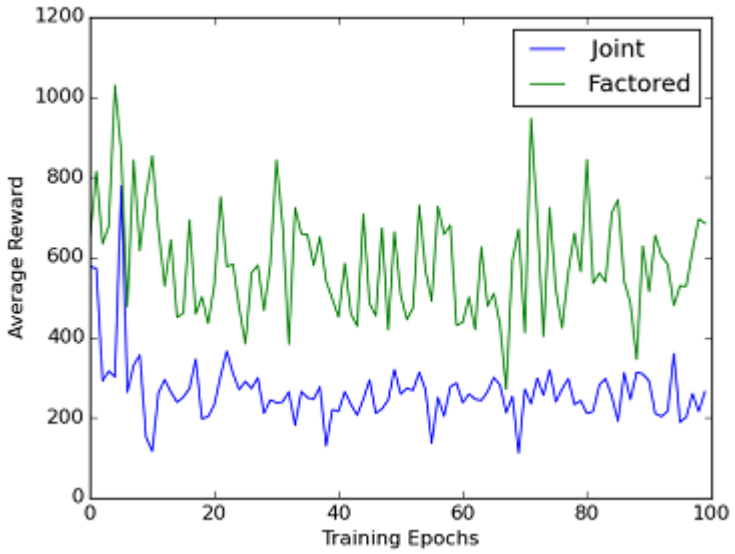
In the factored version, everything is the same except for the final layer, which is replaced with the $3 + 3 + 2 = 8$ outputs needed for the three dimensions. The evaluation, max, and gradient are also changed to match the derivations above. All hyperparameters are unchanged.

Both approaches were tested on the Atari game Asteroids. The reason for choosing this game is that in the factored representation, there is an assumption that every action in the joint action space in meaningful. For some games like Breakout, only a small subset of the 18 actions actually make sense.

Results are summarized in the following three graphs. Joint refers to the control, where the final layer has 18 outputs. Factored refers to the factored representation. Each epoch is 50000 sample actions, divided into several episodes. An episode is one run of the policy, starting from the start state and stopping when the player loses their last life. Loss is averaged across each episode, while mean episode length and average reward are averaged across epochs.


Joint
Factored
Mean Loss
Training Episodes
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
1000
2000
3000
4000
5000
6000
7000
8000


Joint
Factored
Mean Episode Length
Training Epochs
1500
2000
2500
3000
3500
4000
4500
5000
5500
6000
0
20
40
60
80
100



The average loss appears to stay roughly constant over time, but for Q-learning this is not necessarily bad. Recall the loss function changes every time θ changes. Decreasing loss may mean the algorithm plateaus, or it may mean the rate of improvement is constant.

In this case, it unfortunately seems like the former is true. The mean episode length does not increase, much like the control. However, the average reward is immediately much higher than the neural net with the joint action space.

Note however that average reward is immediately much better than the control algorithm. On inspecting a sample episode by the factored agent, its ability to dodge asteroids is qualitatively the same as the control. However, it shoots much more often, earning it more points. This is backed up by the mean episode length being around the same for both approaches.

If the improved performance was just from computational speedup, an improvement of this order of magnitude sounds too good to be true. The different loss curves suggests that the algorithm setup is simply more conducive to this specific problem. In the game of Asteroids, shooting is almost always a good move. In the control algorithm, the shooting decision is spread out over the 9 separate actions describing how the ship should move while shooting. In the factored algorithm, the shooting decision depends on only the 2 actions for that dimension of the action space. The observed reward on shooting is concentrated on just the one output node, which lets it propagate back faster.

Although this domain is a bad example for demonstrating computational speedup, it does suggest that explicitly diving the action space can sometimes give better performance on its own.

6 Further Work

There are other RL approaches that may benefit from applying conditional independence. One promising approach is *policy gradient* methods. In these methods we learn a stochastic policy $\pi(a|s)$, outputting a distribution over actions given the state. A stochastic policy lets us make a smooth update towards improving the policy, allowing us to learn policies directly.

Let a *trajectory* $\tau = (s_0, a_0, s_1, a_1, \cdots)$ be the states and actions seen in one episode. Let θ be the parameters for π. The trajectory depends on the policy, so we can define the following.

$$\ell(\theta) = E_{\tau \leftarrow \pi_\theta}[R(\tau)]$$

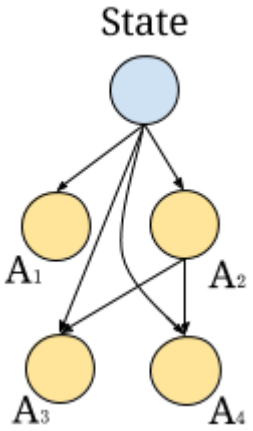
Through some manipulation, we can show the gradient is

$$\nabla_\theta \ell(\tau) = E\left[R(\tau) \sum_t \nabla_\theta \log \pi(a_t|s_t)\right]$$

See [4] for the derivation.

This expectation is still hard to compute exactly, but as an estimate we can sample n trajectories and take the average. Doing gradient ascent will adjust θ towards improved expected reward.

The action distribution can be represented by a directed graphical model like this one, where each action node depends on the state node.



Each action dimension is conditioned on the state and optionally other action dimensions.

Let π_i be the conditional probability at node i. This gives

$$\pi(a|s) = \prod_i \pi_i(a(i)|\,\text{parents}(i), s)$$

To run policy gradient, we need to sample actions a and evaluate $\nabla_\theta \log \pi(a_t|s_t)$. Since the graphical model is a DAG, sampling can be done by choosing the component for the root node, then removing assigned nodes. For the gradient,

$$\nabla_\theta \log \pi(a|s) = \sum_i \nabla_\theta \log \pi_i(a(i)|\,\text{parents}(i), s)$$

which is computable directly from the conditional probabilities.

However, this policy gradient approach is still based on discretizing the continuous action space. It seems that there should be a better approach that exploits the continuous structure. One approach that looks interesting is the wire fitting method [5]. In this approach, the inference algorithm takes state $\boldsymbol{s}$, and outputs pairs (q_i, a_i). The Q-function is then interpolated by these pairs.

$$Q(\boldsymbol{s}, \boldsymbol{a}) = \frac{\sum_i w_i q_i(\boldsymbol{s})}{\sum_i w_i}, \text{ where } w_i \approx 1/\,\text{dist}(\boldsymbol{a}, \boldsymbol{a_i})$$

These pairs are called *wires* because they guide the Q-function. The algorithm is designed such that the wire locations can change based on the state, and those locations depend on the samples received so far. Intuitively, the discretization approach assumes the Q-function has about the same complexity across the entire action space. If the Q-function is complex in a small region (if the Q-function rises and falls many times in that region), then discretization will not model that Q-function well.

The other benefit is that because there is a computable Q-function for an arbitrary continuous action, we can directly sample actions from the continuous space, instead of sampling actions from the discretized space. One approach for turning the Q-function into a policy is to find the maximal action, then add Gaussian noise to the action.

7 Conclusion

Although I unfortunately did not get the approach to work in a continuous problem, the experiments I did do suggests there are merits to this idea, and there are some interesting alternate approaches that are worth trying out.

References

[1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." Nature 518.7540 (2015): 529-533.s

[2] Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. Vol. 1. No. 1. Cambridge: MIT press, 1998.

[3] Guestrin, Carlos, et al. "Efficient solution algorithms for factored MDPs." Journal of Artificial Intelligence Research (2003): 399-468.

[4] Jan Peters (2010) Policy gradient methods. Scholarpedia, 5(11):3698.

[5] Gaskett, Chris, David Wettergreen, and Alexander Zelinsky. "Q-learning in continuous state and action spaces." Australian Joint Conference on Artificial Intelligence. 1999.