Discrete Sequential Prediction of Continuous Actions for Deep RL

Luke Metz*
Google Brain
lmetz@google.com

Julian Ibarz
Google Brain
julianibarz@google.com

Navdeep Jaitly
NVIDIA Research
njaitly@nvidia.com

James Davidson
Google Brain
jcdavidson@google.com

Abstract

It has long been assumed that high dimensional continuous control problems cannot be solved effectively by discretizing individual dimensions of the action space due to the exponentially large number of bins over which policies would have to be learned. In this paper, we draw inspiration from the recent success of sequence-to-sequence models for structured prediction problems to develop policies over discretized spaces. Central to this method is the realization that complex functions over high dimensional spaces can be modeled by neural networks that use next step prediction. Specifically, we show how Q-values and policies over continuous spaces can be modeled using a next step prediction model over discretized dimensions. With this parameterization, it is possible to both leverage the compositional structure of action spaces during learning, as well as compute maxima over action spaces (approximately). On a simple example task we demonstrate empirically that our method can perform global search, which effectively gets around the local optimization issues that plague DDPG and NAF. We apply the technique to off-policy (Q-learning) methods and show that our method can achieve the state-of-the-art for off-policy methods on several continuous control tasks.

1 Introduction

Reinforcement learning has long been considered as a general framework applicable to a broad range of problems. Reinforcement learning algorithms have been categorized in various ways. There is an important distinction, however, arises between discrete and continuous action spaces. In discrete domains, there are several algorithms, such as Q-learning, that leverage backups through Bellman equations and dynamic programming to solve problems effectively. These strategies have led to the use of deep neural networks to learn policies and value functions that can achieve superhuman accuracy in several games [25, 39] where actions lie in discrete domains. This success spurred the development of RL techniques that use deep neural networks for continuous control problems [21, 12, 20]. The gains in these domains, however, have not been as outsized as they have been for discrete action domains, and superhuman performance seems unachievable with current techniques.

This disparity is in part a result of the inherent difficulty in maximizing an arbitrary function on a continuous domain, even in low dimensional settings; further, techniques such as beam searches do not exist or do not apply directly. This makes it difficult to perform inference and learning in such settings. Additionally, it becomes harder to apply dynamic programming methods to back up value function estimates from successor states to parent states. Several of the recent continuous

*Work done as a member of the Google Brain Residency program (g.co/brainresidency)

control reinforcement learning approaches attempt to borrow characteristics from discrete problems by proposing models that allow maximization and backups more easily [12].

One way in which continuous control can avail itself of the above advantages is to discretize each of the dimensions of continuous control action spaces. As noted in [21], doing this naively, however, would create an exponentially large discrete space of actions. For example with M dimensions being discretized into N bins, the problem would balloon to a discrete space with M^N possible actions.

We leverage the recent success of sequence-to-sequence type models [40] to train such discretized models, without falling into the trap of requiring an exponentially large number of actions. Our method relies on a technique that was first introduced in [3], which allows us to escape the curse of dimensionality in high dimensional spaces by modeling complicated probability distributions using the chain rule decomposition. In this paper, we similarly parameterize functions of interest – Q-values – using a decomposition of the joint function into a sequence of conditional approximations. Families of models that exhibit these sequential prediction properties have also been referred to as autoregressive. With this formulation, we are able to achieve fine-grained discretization of individual domains, without an explosion in the number of parameters; at the same time we can model arbitrarily complex distributions while maintaining the ability to perform (approximate) maximization.

While this strategy can be applied to most function approximation settings in RL, we focus on off-policy settings with a DQN inspired algorithm for Q-learning. Complementary results for an actor-critic model [42, 43] based on the same autoregressive concept are presented in Appendix D. Empirical results on an illustrative multimodal problem demonstrates how our model is able to perform global maximization, avoiding the exploration problems faced by algorithms like NAF [13] and DDPG [21]. We also show the effectiveness of our method in solving a range of benchmark continuous control problems including hopper to humanoid.

2 Method

In this paper, we introduce the idea of building continuous control algorithms utilizing sequential, or autoregressive, models that predict over action spaces one dimension at a time. Here, we use discrete distributions over each dimension (achieved by discretizing each continuous dimension into bins) and apply it on off-policy learning. We explore one instantiation of such a model in the body of this work and discuss three additional variants in Appendix C, Appendix D and Appendix E.

2.1 Preliminaries

We briefly describe the notation we use in this paper. Let $\boldsymbol{s}_t \in \mathbb{R}^L$ be the observed state of the agent, $\boldsymbol{a} \in \mathbb{R}^N$ be the N dimensional action space, and $\mathcal{E}$ be the stochastic environment in which the agent operates. Finally, let $\boldsymbol{a}^{i:j} = \left[a^i \cdots a^j\right]^T$ be the vector obtained by taking the sub-range of a vector $\boldsymbol{a} = \left[a^1 \cdots a^N\right]^T$.

At each step t, the agent takes an action $\boldsymbol{a}_t$, receives a reward r_t from the environment and transitions stochastically to a new state $\boldsymbol{s}_{t+1}$ according to (possibly unknown) dynamics $p_{\mathcal{E}}(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$. An episode consists of a sequence of such steps $(\boldsymbol{s}_t, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$, with $t = 1 \cdots H$ where H is the last time step. An episode terminates when a stopping criterion $F(\boldsymbol{s}_{t+1})$ is true (for example when a game is lost, or when the number of steps is greater than some threshold length H_{max}).

Let $R_t = \sum_{i=t}^{H} \gamma^{i-1} r_i$ be the discounted reward received by the agent starting at step t of an episode. As with standard reinforcement learning, the goal of our agent is to learn a policy $\pi(\boldsymbol{s}_t)$ that maximizes the expected total reward $\mathbb{E}[R_H]$ it would receive from the environment by following this policy.

Because this paper is focused on off-policy learning with Q-Learning [48], we will provide a brief description of the algorithm.

2.1.1 Q-Learning

Q-learning is an off-policy algorithm that learns an action-value function $Q(\boldsymbol{s}, \boldsymbol{a})$ and a corresponding greedy-policy, $\pi^Q(\mathbf{s}) = \text{argmax}_{\boldsymbol{a}} Q(\boldsymbol{s}, \boldsymbol{a})$. The model is trained by finding the fixed point of the

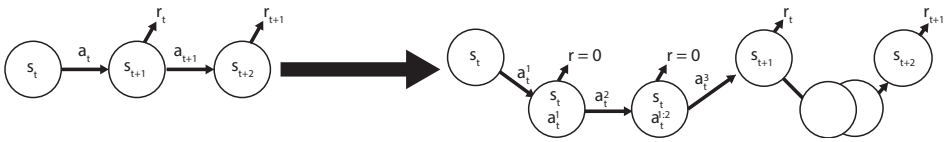

Figure 1: Demonstration of a transformation on an environment with three dimensional action space. Fictitious states are introduced to keep the action dimension at each transition one dimensional. Each circle represents a state in the MDP. The transformed environment's replicated states are now augmented with the previously selected action. When all three action dimensions are chosen, the underlying environment progresses to $\boldsymbol{s}_{t+1}$.

transition operator, i.e.

$$Q(\boldsymbol{s_t}, \boldsymbol{a_t}) = \mathbb{E}_{\boldsymbol{s_{t+1}} \sim p_{\mathcal{E}}(\cdot|\boldsymbol{s_t}, \boldsymbol{a_t})}[r + \gamma Q(\boldsymbol{s_{t+1}}, \pi^Q(\boldsymbol{s_{t+1}}))] \qquad \forall(\boldsymbol{s_t}, \boldsymbol{a_t}) \tag{1}$$

This is done by minimizing the Bellman Error, over the exploration distribution, $\rho_\beta(\boldsymbol{s})$

$$L = \mathbb{E}_{\boldsymbol{s_t} \sim \rho_\beta(\cdot), \boldsymbol{s_{t+1}} \sim \rho_{\mathcal{E}}(\cdot|\boldsymbol{s_t}, \boldsymbol{a_t})} \| Q(\boldsymbol{s_t}, \boldsymbol{a_t}) - (r + \gamma Q(\boldsymbol{s_{t+1}}, \pi^Q(\boldsymbol{s_{t+1}}))) \|^2 \tag{2}$$

Traditionally, Q is represented as a table of state action pairs or with linear function approximators or shallow neural networks [48, 45]. Recently, there has been an effort to apply these techniques to more complex domains using non-linear function approximators that are deep neural networks [25, 26]. In these models, a *Deep Q-Network* (DQN) parameterized by parameters, θ, is used to predict Q-values, i.e. $Q(\boldsymbol{s}, \boldsymbol{a}) = f(\boldsymbol{s}, \boldsymbol{a}; \theta)$. The DQN parameters, θ, are trained by performing gradient descent on the error in equation 2, without taking a gradient through the Q-values of the successor states (although, see [2] for an approach that takes this into account).

Since the greedy policy, $\pi^Q(\boldsymbol{s})$, uses the action value with the maximum Q-value, it is essential that any parametric form of Q be able to find a maxima easily with respect to actions. For a DQN where the output layer predicts the Q-values for each of the discrete outputs, it is easy to find this max – it is simply the action corresponding to the index of the output with the highest estimated Q-value. In continuous action problems, it can be tricky to formulate a parametric form of the Q-value where it is easy to find such a maxima. Techniques like Normalized Advantage Function (NAF)[13] constrain the function approximator such that it is easy to compute a max analytically at the cost of reduced flexibility. Another approach has been followed by the Deep Deterministic Policy Gradients algorithm, which uses an actor-critic algorithm in an off-policy setting to train a deterministic policy [21]. The model trains a critic to estimate the Q-function and a separate policy to try to approximate a max over the critic.

In this work, we develop a similar technique where we modify the form of our Q-value function while still retaining the ability to find local maxima over actions for use in a greedy policy.

2.2 Sequential DQN

We propose a Sequential DQN (SDQN) model that structures a Q function in the form of a sequential model over action dimensions. To introduce this idea, we first show a transformation done on the environment which splits a ND action into a sequence of 1D actions. We then shift this modification from an environment transformation to the construction of a Q function resulting in our final technique.

2.2.1 Transformed Environment

Consider an environment with an N dimensional action space. We can perform a transformation to this environment into a similar environment inserting $N - 1$ fictitious states. Each new fictitious state has a 1D action space. Each new action choice is conditioned on the previous state as well as on all previously selected actions. When the Nth action is selected, the environment dynamics take into effect using the previous N actions to perform one step. The process is depicted for $N = 3$ in Figure 1.

This transformation reduces the ND actions to a series of 1D actions. We can now discretize the 1D output space and directly apply Q-learning. Note that we could apply this strategy to continuous

values, without discretization, by choosing a conditional distribution, such as a mixture of 1-D Gaussians, over which a maxima can easily be found. As such, this approach is equally applicable to pure continuous domains as compared to discrete approximations.

The downside to this transformation is that it increases the amount of steps needed to solve a MDP. In practice, this transformation makes learning a Q function considerably harder. The extra steps of dynamic programming coupled with learned function approximators causes large overestimation and stability issues.

2.2.2 Model

In this section, we shift the previous section's ideas into a model that acts on the original, untransformed environment. We learn a deterministic policy $\pi(\boldsymbol{s})$ defined as:

$$\pi(\boldsymbol{s}) = [\pi^1(\boldsymbol{s}), \pi^2(\boldsymbol{s}), .., \pi^N(\boldsymbol{s})]. \tag{3}$$

Here N is the number of action dimensions, and $\pi^i(\boldsymbol{s})$ is the deterministic policy for the i dimension action choice, i.e. $a^i = \pi^i(\boldsymbol{s})$. A greedy policy from Q-learning is used to compute these policies:

$$\pi^i(\boldsymbol{s}) = \underset{a^i \in \mathcal{A}^i}{\operatorname{argmax}} Q^i(\boldsymbol{s}, [\pi^{1:i-1}(\boldsymbol{s}), a^i]), \tag{4}$$

where $\pi^{j:i}(\boldsymbol{s}) = [\pi^j(\boldsymbol{s}), ..., \pi^i(\boldsymbol{s})]$ and $\mathcal{A}^i$ is the set of possible actions for values for a^i. Here $Q^i(s, a^{1:i})$ is a "partial Q-value function" that we estimate using Q-learning and is defined recursively because $\pi^{1:i-1}$ is an input to Q^i. These "partial Q-value functions" correspond to the Q-values occurring on the fictitious states introduced in the previous section.

We parameterize $Q^i(\boldsymbol{s}, a^{1:i})$ as a neural network that receives $(\boldsymbol{s}, \pi^{1:i-1}(\boldsymbol{s}))$ as input. The neural network outputs a B dimensional vector, where B is the number of bins that we discretized action space $\mathcal{A}^i$ into. This parameterization allows for arbitrarily complex resolution at the cost of increasing the number of bins, B. We found $B = 32$ to be a good tradeoff between performance and tractability.

This model is similar to doing Q-learning on the transformed environment described in the previous section. As before, the entire chain of Q^i 's can then be trained with Q-learning. In practice, however, we found that the Q values were significantly overestimated, due to the increased time dependencies and the max operator in Q-learning. When training, there is an approximation error in Q^i value predictions. This approximation error and a max operator in Q-learning causes overestimation. To overcome this issue we introduce a second function, $Q^D(\boldsymbol{s}, \boldsymbol{a}) \in \mathbb{R}$, inspired by double DQN [15], which we use to produce a final estimate of the Q-value of all action dimensions. Note, in the traditional double DQN work, a past version of the Q function is used. In our setting, we train an entirely separate network, Q^D, on the original, untransformed environment. By doing this, learning Q^D becomes easier as there are less long term dependencies.

There are several components we must train for our model. First, Q^D minimizes the Bellman Error:

$$l_{td} = \mathbb{E}_{(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{s}_{t+1}) \in R}[(r + \gamma Q^D(\boldsymbol{s}_{t+1}, \pi^N(\boldsymbol{s}_{t+1})) - Q^D(\boldsymbol{s}_t, \boldsymbol{a}_t))^2]. \tag{5}$$

Next, Q^N is trained to match the predicted value from Q^D:

$$l_{base} = \mathbb{E}_{\boldsymbol{s} \in R}[(Q^D(\boldsymbol{s}, \pi^N(\boldsymbol{s})) - Q^N(\boldsymbol{s}, \pi^N(\boldsymbol{s})))^2]. \tag{6}$$

Finally, Q^i learns to satisfy the Bellman equation ensuring that $Q^i(\boldsymbol{s}, [a^1 \cdots a^i]) \approx \max_x Q^{i+1}(\boldsymbol{s}, [a^1 \cdots a^i, x])$. We do this by creating an internal consistency loss for each dimension i:

$$l^i_{inner} = \mathbb{E}_{(\boldsymbol{s}, \boldsymbol{a}) \in R} \| Q^i(\boldsymbol{s}, \boldsymbol{a}^{1:i}) - \max_{a^{i+1} \in \mathcal{A}^{i+1}} Q^{i+1}(\boldsymbol{s}, [\boldsymbol{a}^{1:i}, a^{i+1}]) \|^2. \tag{7}$$

We sample $(\boldsymbol{s}_{t+1}, \boldsymbol{a}_t, r_t, \boldsymbol{s}_{t+1})$ transitions from a replay buffer, R, to minimize all three losses using SGD [25]. As in [25], at each training step, R is filled with one transition sampled using the current policy, π.

The policy action selection, π, is illustrated in Figure 2, while training is depicted in Figure A.1.

Another motivation for structuring our losses in this way is to decouple the model that learns Q values from the model that chooses max actions – or the policy. In this way our approach is similar to DDPG. A Q value is learned in one network, Q^D, and a policy to approximate the max is learned in another, Q^i.

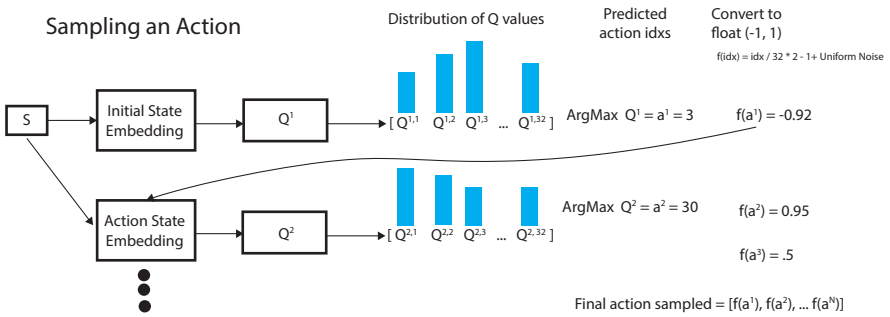

Figure 2: Pictorial view of the SDN policy, π, with 32 discretization bins. A state, s, is fed into the network, gets embedded, and a distribution of Qs for the first action dimension, a^1, are predicted. The max bin is selected, and converted to a continuous action. This action is then fed in with the state to predict the Q distribution of the second action dimension, a^2. See Figure A.1 for a pictorial view off the training procedure.

2.3 Exploration

Numerous exploration strategies have been explored in reinforcement learning [32, 5, 17]. When computing actions from our sequential policies, we can additionally inject noise at each choice and then select the remaining actions based on this choice. We found this yields better performance as compared to epsilon greedy.

2.4 Neural Network Parameterization

We implemented each learned function in the above model as a deep neural network. We explored two parameterizations of the sequential components, the Q^is. First, we looked at a recurrent LSTM model [16]. This model has shared weights and passes information via hidden activations from one action to another. The input at each time step is a function of the current state, $\boldsymbol{s}$, and action, a^i. Second, we looked at a version with no shared weights, passing all information into each sequential prediction model. These models are feed forward neural networks that take as input a concatenation of all previous action selections as well as the state. In more complex domains, such as vision based control tasks for example, it would make sense to untie only a subset of the weights. In practice, we found that the untied version performed better. Optimizing these model families is still an ongoing work. For full detail of model architectures and training procedures selection see Appendix F.

3 Related Work

Our work was motivated by two distinct desires – to learn policies over exponentially large discrete action spaces, and to approximate value functions over high dimensional continuous action spaces effectively. In our paper we used a sequential parameterization of policies that help us to achieve this without making an assumption about the actual functional form of the model. Other prior work attempts to handle high dimensional action spaces by assuming specific decompositions. For example, [36] were able to scale up learning to extremely large action sizes by factoring the action value function and use product of experts to learn policies. An alternative strategy was proposed in [11] using action embeddings and applying k-nearest neighbors to reduce scaling of action sizes. By laying out actions on a hypercube, [33] are able to perform a binary search over actions resulting in a logarithmic search for the optimal action. Their method is similar to SDQN, as both construct a Q-value from sub Q-values. Their approach presupposes these constraints, however, and optimizes the Bellman equation by optimizing hyperplanes independently thus enabling optimizing via linear programming. Our approach is iterative and refines the action selection, which contrasts to their independent sub-plane maximization.

Along with the development of discrete space algorithms, researchers have innovated specialized solutions to learn over continuous state and action environments including [19, 21, 13]. More recently, novel deep RL approaches have been developed for continuous state and action problems. TRPO [37] and A3C [24] uses a stocastic policy parameterized by diagonal covariance Gaussian distributions.

NAF [13] relies on quadratic advantage function enabling closed form optimization of the optimal action. Other methods structure the network in a way such that they are convex in the actions while being non-convex with respect to states [1] or use a linear policy [34].

In the context of reinforcement learning, sequential or autoregressive policies have previously been used to describe exponentially large action spaces such as the space of neural architectures, [50] and over sequences of words [30, 38]. These approaches rely on policy gradient methods whereas we explore off-policy methods. Hierarchical/options based methods, including [9] which perform spatial abstraction or [44] that perform temporal abstraction pose another way to factor action spaces. These methods refine their action selection from time where our approaches operates on the same timescale and factors the action space.

A vast literature on constructing sequential models to solve tasks exists outside of RL. These models are a natural fit when the data is generated in a sequential process such as in language modeling [4]. One of the first and most effective deep learned sequence-to-sequence models for language modeling was proposed in [41], which used an encoder-decoder architecture. In other domains, techniques such as NADE [18] have been developed to compute tractable likelihood. Techniques like Pixel RNN [31] have been used to great success in the image domain where there is no clear generation sequence. Hierarchical softmax [27] performs a hierarchical decomposition based on WordNet semantic information.

The second motivation of our work was to enable learning over more flexible, possibly multimodal policy landscape. Existing methods use stochastic neural networks [7] or construct energy models [14] sampled with Stein variational gradient descent [22, 47]. In our work instead of sampling, we construct a secondary network to evaluate a max.

4 Experiments

4.1 Multimodal Example Environment

To consider the effectiveness of our algorithm, we consider a deterministic environment with a single time step, and a 2D action space. This can be thought of as being a two-armed bandit problem with deterministic rewards, or as a search problem in 2D action space. We chose our reward function to be a multimodal distribution as shown in the first column in Figure 3. A large suboptimal mode and a smaller optimal mode exist.

As with bandit problems, this formulation helps us isolate the ability of our method to find an optimal policy, without the confounding effect that arises from backing up rewards via the Bellman operator for sequential problems. We look at the behavior of SDQN as well as that of DDPG and NAF on this task. As in traditional RL, we do exploration while learning. We consider uniformly sampling (ϵ-greedy with $\epsilon = 1$) as well as sampling data from a normal distribution centered at the current policy – we refer to this as "local." A visualization of the final Q surfaces as well as training curves can be found in Figure 3.

First, we considered the performance of DDPG. DDPG uses local optimization to learn a policy on a constantly changing estimate of Q values predicted by a critic. The form of the Q distribution is flexible and as such there is no closed form properties we can make use of for learning a policy. As such, we resort to gradient descent, a local optimization algorithm. Due to its local nature, it is possible for this algorithm to get stuck in sub-optimal policies that are local maximum of the current critic. We hypothesize that these local maximum in policy space exist in more realistic simulated environments as well. Traditionally, deep learning methods use local optimizers and avoid local minima or maxima by working in a high dimensional parameter space [8]. In RL, however, the action space of a policy is relatively small dimensionally thus it is much more likely that they exist. For example, in the hopper environment, a common failure mode we experienced when training algorithms like DDPG is to learn to balance instead of moving forward and hopping.

Next, we consider how NAF behaves on this environment. NAF makes an assumption that the Q function is quadratic in action space. During training, NAF fits this quadratic surface to minimize the expected ℓ_2 loss evaluated with transitions from a replay buffer. Given the restricted functional form of the model, it is no longer possible to model the entire space without error. As such, the distribution of sampled points used for learning (i.e. the behavior policy) matters greatly. When the behavior policy is a uniform distribution over the action space, the quadratic approximation yields a surface

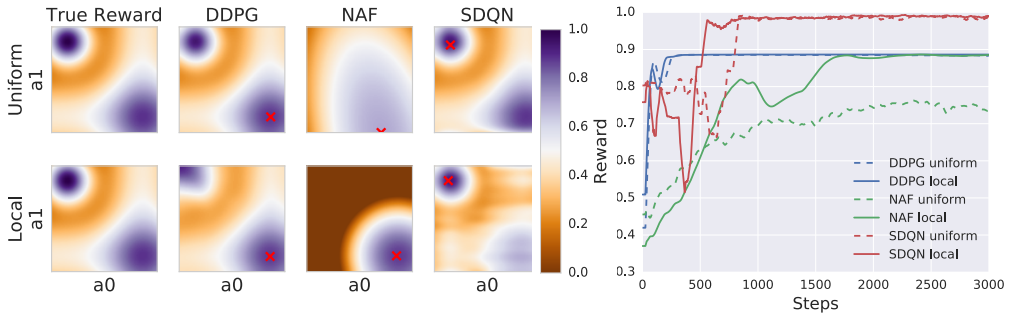

Figure 3: Left: Final reward/Q surface for each algorithm tested. Final policy is marked with a red $\times$. The SDQN model is capable of performing global search and thus finds the global maximum. The top row contains data collected uniformly over the action space. SDQN and DDPG use this to accurately reconstruct the target Q surface. Algorithms like NAF, however, fail to even converge to a local maximum. In the bottom row, actions are sampled from a normal distribution centered on the policy. This results in more sample efficiency but yields poor approximations of the Q surface outside of where the policy is. Right: Smoothed reward achieved over time. DDPG quickly converges to a local maximum. SDQN has high variance performance initially as it searches the space, but then quickly converges to the global maximum as the Q surface estimate becomes more accurate. NAF, when sampling uniformly, fails to converge to a global maximum. The location of the max is actually lower than a global maximum.

where the maximum is in a low reward region with no path to improve. Interestingly though, this is no longer the case when the behavior policy is a stochastic Gaussian policy whose mean is the greedy policy w.r.t. the previous estimate of the Q values. In this setting, NAF only models the quadratic surface around where samples are taken, i.e. locally around the maxima of the estimated Q values. This non-stationary optimization results in behavior quite similar to DDPG in that it performs a local optimization. This experiment suggests that in NAF there is a balance between a global quadratic model, and exploiting local structure when fitting a Q function.

Finally, we consider a SDQN. As expected, this model is capable of completely representing the Q surface (under the limits of discretization) and does not suffer from inflexibility of the previous methods. The optimization of the policy is not done locally – both the uniform behavior policy and the stochastic Gaussian behavior policy converge to the optimal solution. Much like DDPG, the loss surface learned can be stationary – it does not need to shift over time to learn the optimal solution. Unlike DDPG, however, the policy will not get stuck in a local maximum. With the uniform behavior policy setting, the model slowly reaches the right solution, as there are many wasted samples. With a behavior policy that is closer to being on-policy (such as the stochastic Gaussian greedy policy referred to above), this slowness is reduced. Much of the error occurs from selecting over estimated actions. When sampling more on policy, the over estimated data points get sampled more frequently converging to the optimal solution. Unlike the other global algorithms like NAF, performance will not decrease if we increase sampling noise to cover more of the action space. This allows us to balance exploration and learning of the Q function more easily. [2]

4.2 Mujoco environments

To evaluate the relative performance of these models we perform a series of experiments on common continuous control tasks. We test the hopper, swimmer, half cheetah, walker2d and humanoid environments from the OpenAI gym suite [6]. [3]

We performed a wide hyper parameter search over various parameters in our models (described in Appendix F), and selected the best performing runs. We then ran 10 random seeds of the same hyper parameters to evaluate consistency and to get a more realistic estimate of performance. We

[2] This assumes that the models have enough capacity. In a limited capacity setting, one would still want to explore locally. Much like NAF, SDQN models will shift capacity to modeling these spaces, which are sampled, thus making better use of the capacity.

[3] For technical reasons, our simulations use a different numerical simulation strategy provided by Mujoco [46]. In practice though, we found the differences in final reward to be within the expected variability of rerunning an algorithm with a different random seed.

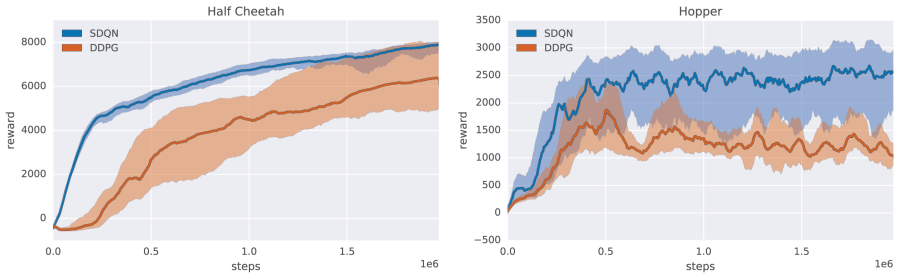

Figure 4: Learning curves of highest performing hyper parameters trained on Mujoco tasks. We show a smoothed median (solid line) with 25 and 75 percentiles range (transparent line) from the 10 random seeds run. SDQN quickly achieves good performance on these tasks.

agent	hopper	swimmer	half cheetah	humanoid	walker2d
SDQN	**3342.62**	**179.23**	**7774.77**	**3096.71**	3227.73
DDPG	3296.49	133.52	6614.26	3055.98	**3640.93**

Figure 5: Maximum reward achieved over training averaged over a 25,000 step window with evaluations every 5,000 steps. Results are averaged over 10 randomly initialized trials with fixed hyper parameters. SDQN models perform competitively as compared to DDPG.

believe this replication is necessary as many of these algorithms are not only sensitive to both hyper parameters but random seeds. This is not a quality we would like in our RL algorithms and by doing the 10x replications, we are able to detect this phenomena.

First, we look at learning curves of some of the environments tested in Figure 4. Our method quickly achieves good policies much faster than DDPG.

Next, for a more qualitative analysis, we use the best reward achieved while training averaged across over 25,000 steps and with evaluations sampled every 5,000 steps. Again we perform an average over 10 different random seeds. This metric gives a much better sense of stability than the traditionally reported instantaneous max reward achieved during training.

We compare our algorithm to the current state-of-the-art in off-policy continuous control: DDPG. Through careful model selection and extensive hyper parameter tuning, we train models with performance better than previously published for DDPG on some of these tasks. Despite this search, however, we believe that there is still space for *significant* performance gain for all the models given different neural network architectures and hyper parameters. Results can be seen in Figure 5. Our algorithm achieves better performance on four of the five environments we tested.

5 Discussion

Conceptually, our approach centers on the idea that action selection at each stage can be factored and sequentially selected using an autoregressive formulation. In this work we use 1D action spaces that are discretized. Existing work in the image modeling domain suggests that using a mixture of logistic units [35] greatly speeds up training and would also satisfy our need for a closed form max. Additionally, this work imposes a prespecified ordering of actions which may negatively impact training for certain classes of problems. To address this, we could learn to factor the action space into the sequential order for continuous action spaces or learn to group action sets for discrete action spaces. Another promising direction is to combine this approximate max action with gradient based optimization procedure. This would relieve some of the complexity of the modeling task of the maxing network, at the cost of increased compute when sampling from the policy. Finally, the work presented here is exclusively on off-policy methods. Use of an autoregressive policy with discretized actions could also be used as the policy for any stochastic policy optimization algorithm such as TRPO [37] or A3C [24].

6 Conclusion

In this work we present a continuous control algorithm that utilize discretized action spaces and sequential models. The technique we propose is an off-policy RL algorithm that utilizes sequential prediction and discretization. We decompose our model into a Q function and an auxiliary network that acts as a policy and is responsible for computing an approximate max over actions. The effectiveness of our method is demonstrated on illustrative and benchmark tasks, as well as on more complex continuous control tasks. Two additional formulations of discretized sequential prediction models are presented in Appendix C and Appendix D

Acknowledgements

We would like to thank Nicolas Heess for his insight on exploration and assistance in scaling up task complexity, and Oscar Ramirez for his assistance running some experiments. We would like to thank Eric Jang, Sergey Levine, Mohammad Norouzi, Leslie Phillips, Chase Roberts, and Vincent Vanhoucke for their comments and feedback. Finally we would like to thank the entire Brain Team for their support.

References

[1] Brandon Amos, Lei Xu, and J Zico Kolter. Input convex neural networks. *arXiv preprint arXiv:1609.07152*, 2016.

[2] Leemon Baird. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, pages 30–37. Morgan Kaufmann, 1995.

[3] Yoshua Bengio and Samy Bengio. Modeling high-dimensional discrete data with multi-layer neural networks.

[4] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb):1137–1155, 2003.

[5] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. *arXiv preprint arXiv:1505.05424*, 2015.

[6] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[7] Pieter Abbeel Carlos Florensa, Yan Duan. Stochastic neural networks for hierarchical reinforcement learning. In *International Conference on Learning Representations*, 2017.

[8] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *AISTATS*, 2015.

[9] Peter Dayan and Geoffrey E Hinton. Feudal reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 271–271. Morgan Kaufmann Publishers, 1993.

[10] Thomas Degris, Martha White, and Richard S. Sutton. Off-policy actor-critic. *CoRR*, abs/1205.4839, 2012.

[11] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. Deep reinforcement learning in large discrete action spaces. *arXiv preprint arXiv:1512.07679*, 2015.

[12] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. *arXiv preprint arXiv:1603.00748*, 2016.

[13] Shixiang Gu, Timothy P. Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. *CoRR*, abs/1603.00748, 2016.

[14] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies. *arXiv preprint arXiv:1702.08165*, 2017.

[15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100. AAAI Press, 2016.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[17] Rein Houthooft, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. Vime: Variational information maximizing exploration. In *Advances in Neural Information Processing Systems*, pages 1109–1117, 2016.

[18] Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *AISTATS*, volume 1, page 2, 2011.

[19] Guy Lever. Deterministic policy gradient algorithms. 2014.

[20] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[21] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[22] Qiang Liu and Dilin Wang. Stein variational gradient descent: A general purpose bayesian inference algorithm. In *Advances In Neural Information Processing Systems*, pages 2370–2378, 2016.

[23] Andriy Mnih and Danilo J Rezende. Variational inference for monte carlo objectives. *arXiv preprint arXiv:1602.06725*, 2016.

[24] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy P Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, 2016.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[26] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[27] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[28] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc Bellemare. Safe and efficient off-policy reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1046–1054, 2016.

[29] Ofir Nachum, Mohammad Norouzi, Kelvin Xu, and Dale Schuurmans. Bridging the gap between value and policy based reinforcement learning. *arXiv preprint arXiv:1702.08892*, 2017.

[30] Mohammad Norouzi, Samy Bengio, Zhifeng Chen, Navdeep Jaitly, Mike Schuster, Yonghui Wu, and Dale Schuurmans. Reward augmented maximum likelihood for neural structured prediction. In *Neural Information Processing Systems (NIPS)*, 2016.

[31] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

[32] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. Deep exploration via bootstrapped dqn. In *Advances In Neural Information Processing Systems*, pages 4026–4034, 2016.

[33] Jason Pazis and Ron Parr. Generalized value functions for large action sets. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1185–1192, 2011.

[34] Aravind Rajeswaran, Kendall Lowrey, Emanuel Todorov, and Sham Kakade. Towards generalization and simplicity in continuous control. *arXiv preprint arXiv:1703.02660*, 2017.

[35] Tim Salimans, Andrej Karpathy, Xi Chen, and Diederik P Kingma. Pixelcnn++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*, 2017.

[36] Brian Sallans and Geoffrey E Hinton. Reinforcement learning with factored states and actions. *Journal of Machine Learning Research*, 5(Aug):1063–1088, 2004.

[37] John Schulman, Sergey Levine, Pieter Abbeel, Michael I Jordan, and Philipp Moritz. Trust region policy optimization. In *ICML*, pages 1889–1897, 2015.

[38] Shiqi Shen, Yong Cheng, Zhongjun He, Wei He, Hua Wu, Maosong Sun, and Yang Liu. Minimum risk training for neural machine translation. *arXiv preprint arXiv:1512.02433*, 2015.

[39] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[40] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[41] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[42] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[43] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, volume 99, pages 1057–1063, 1999.

[44] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[45] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[46] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, pages 5026–5033. IEEE, 2012.

[47] Dilin Wang and Qiang Liu. Learning to draw samples: With application to amortized mle for generative adversarial learning. *arXiv preprint arXiv:1611.01722*, 2016.

[48] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[49] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[50] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

Appendix

A Model Diagrams

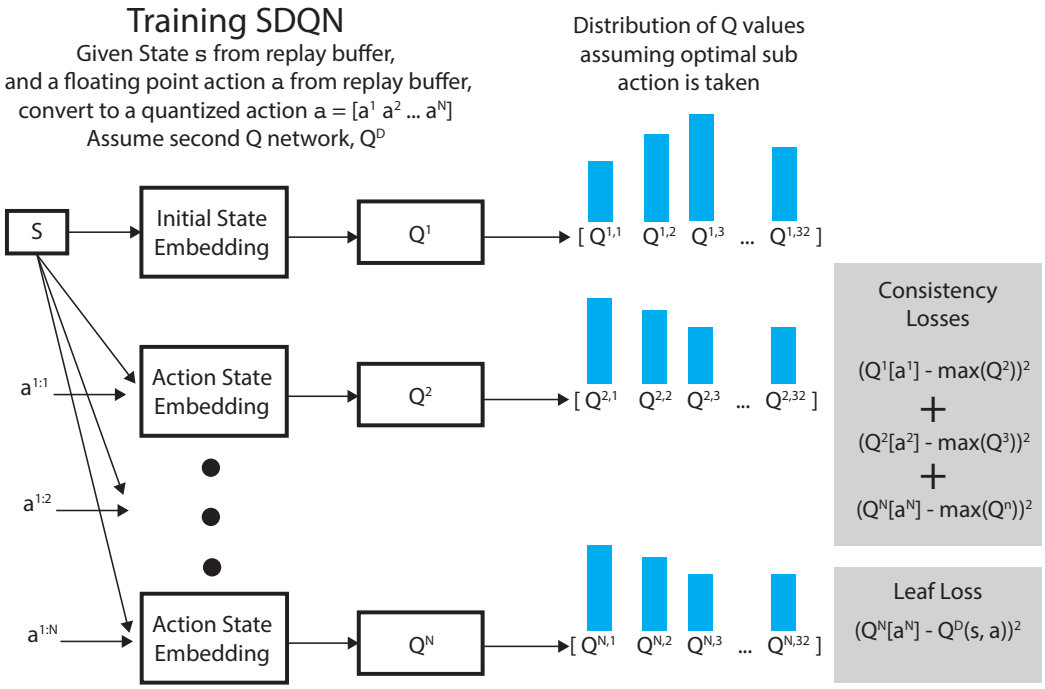

Figure A.1: Pictorial view for the SDQN network showing training. See Figure 2 for model in evaluation mode.

B Model Visualization

To gain insight into the characteristics of Q that our SDQN algorithm learns, we visualized results from the hopper environment, because these are easier to comprehend.

First we compute each action dimension's Q distribution, Q^i, and compare those distributions to that of the double DQN network for the full action taken, Q^D. A figure containing these distributions and corresponding state visualization can be found in Figure B.2.

For most states in the hopper walk cycle, the Q distribution is very flat. This implies that small changes in the action taken in a state will have little impact on future reward. This makes sense as the system can recover from any action taken in this frame. However, this is not true for all states – certain critical states exist, such as when the hopper is pushing off, where not selecting the correct action value greatly degrades performance. This can be seen in frame 466.

Our algorithm is trained with a number of soft constraints. First, if fully converged, we would expect $Q^i >= Q^{i-1}$ as every new sub-action taken should maintain or improve the expected future discounted reward. Additionally, we would expect $Q^N(s, a) = Q^D(s, a)$ (from eq. 2.2.2). In the majority of frames these properties seem correct, but there is certainly room for improvement.

Next, we attempt to look at Q surfaces in a more global manner. We plot 2D cross sections for each pair of actions and assume the third dimension is zero. Figure B.3 shows the results.

As seen in the previous visualization, the surface of both the autoregressive Q surface and the Q^D is not smooth, which is expected as the environment action space for Hopper is expected to be highly

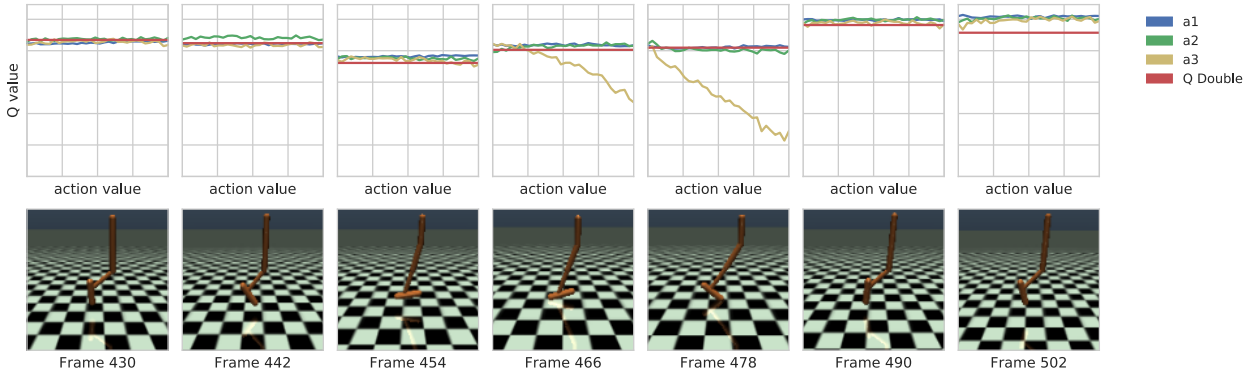

Figure B.2: Exploration of the sub-DQN during after training. The top row shows the Q^i predictions for a given frame (action dimensions correspond to the joint starting at the top and moving toward the bottom – action 3 is the ankle joint). The bottom row shows the corresponding rendering of the current state. For insensitive parts of the gait, such as when the hopper is in the air (e.g. frame 430, 442, 490, 502), the network learns to be agnostic to the choice of actions; this is reflected in the flat Q-value distribution, viewed as a function of action index. On the other hand, for critical parts of the gait, such as when the hopper is in contact with the ground (e.g. frames 446, 478), the network learns that certain actions are much better than others, and the Q-distribution is no longer a flat line. This reflects the fact that taking wrong actions in these regimes could lead to bad results such as tripping, yielding a lower reward.

non-linear. Some regions of the surface seem quite noisy which is not expected. Interestingly though, these regions of noise do not seem to lower the performance of the final policy. In Q-learning, only the maximum Q value regions have any impact on the taken policy. Future work is needed to better characterize this effect. We would like to explore techniques that use "soft" Q-learning [29, **?**, 14]. These techniques will use more of the Q surface thus smooth the representations.

Additionally, we notice that the dimensions of the autoregressive model are modeled differently. The last action, a_3 has considerably more noise than the previous two action dimensions. This large difference in the smoothness and shape of the surfaces demonstrates that the order of the actions dimensions matters. This figure suggests that the model has a harder time learning sharp features in the a_1 dimension. In future work, we would like to explore learned orderings, or bidirectional models, to combat this.

Finally, the form of Q^D is extremely noisy and has many cube artifacts. The input of this function is both a one hot quantized action, as well as the floating point representation. It appears the model uses the quantization as its main feature and learns a sharp Q surface.

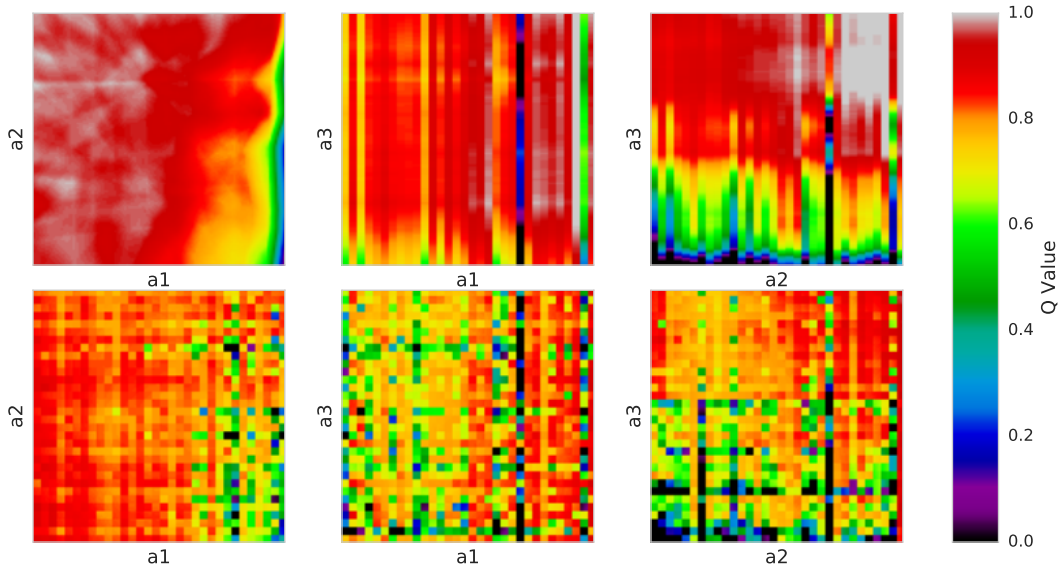

Figure B.3: Q surfaces given a fixed state. Top row is the autoregressive model, Q^N. The bottom row is the double DQN, Q^D. We observe high noise in both models. Additionally, we see smoother variation in earlier action dimensions, which suggests that conditioning order matters greatly. Q values are computed with a reward scale of 0.1, and a discounted return of 0.995.

C Add SDQN

In this section, we discuss a different model that also makes use of sequential prediction and quantization. The SDQN model uses partial-DQN's, $Q^i(\cdot, \cdot)$, to define sequential greedy policies over all dimensions, i. In this setting, one can think of it as acting similarly to an environment transformed to predict one dimension at a time. Thus reward signals from the final reward must be propagated back through a chain of partial-DQN. This results in a more difficult credit assignment problem that needs to be solved. This model attempts to solve this by changing the structure of Q networks. This formulation, called *Add SDQN* replaces the series of maxes from the Bellman backup with a summation over learned functions.

Results for this method and the others presented in the appendices can be found in Appendix E.2.

C.1 Method

As before, we aim to learn a deterministic policy $\pi(\boldsymbol{s})$ of the DQN, where

$$\pi(\boldsymbol{s}) = \underset{a}{\operatorname{argmax}}\, Q(\boldsymbol{s}, \boldsymbol{a}) \tag{8}$$

Here the Q value is defined as the sum of F:

$$Q(\boldsymbol{s}, \boldsymbol{a}) = \sum_{i=1}^{N} F^i(\boldsymbol{s}, \boldsymbol{a}^{1:i}) \tag{9}$$

Unlike before, the sequential components no longer represent Q functions, so we will swap the notation of our compositional function from Q^i to F^i.

The parameters of all the F^i models are trained by matching $Q(\cdot, \cdot)$, in equation 9 to Q^D as follows:

$$l_{matching} = \mathbb{E}_{\boldsymbol{s} \in R}[(Q^D(\boldsymbol{s}, \pi^N(\boldsymbol{s})) - Q(\boldsymbol{s}, \pi^N(\boldsymbol{s})))^2] \tag{10}$$

We train Q with Q-learning, as shown in equation 5.

Unlike the sequential max policy of previous section, here we find the optimal action by beam search to maximize equation 9. In practice, the learning dynamics of the neural network parameterizations we use yield solutions that are amenable to this shallow search and do not require a search of the full exponential space.

A figure showing this network's training procedure can be found in Figure C.4.

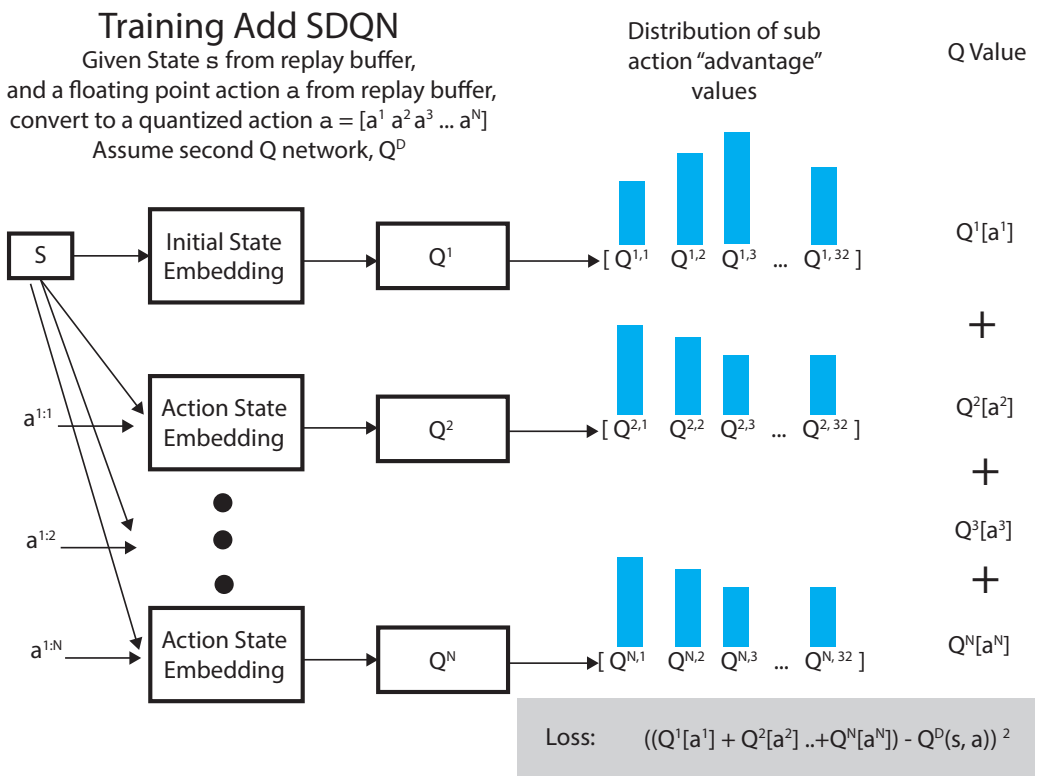

Figure C.4: Pictorial view for the Add network showing both training. Policy evaluation from this network is done in a procedure similar to that shown in Figure 2

C.2 Network Parameterization Note

At this point, we have only tested the LSTM variant and not the untied parameterization. Optimizing these model families is an ongoing work and we could assume that Add SDQN could potentially perform better if it were using the untied version as we did for the originally presented SDQN algorithm.

D Prob SDQN

In the previous sections we showed the use of our technique within the Q-learning framework. Here we describe its use within the off-policy, actor-critic framework to learn a sequential policy [42, 43, 10]. We name this model *Prob SDQN*.

Results for this method and the others presented in the appendices can be found in Appendix E.2.

D.1 Method

We define a policy, π, recursively as follows $\pi(s) = \pi^N(s)$ where π^i is defined as:

$$\pi(\boldsymbol{s}) = [\pi^1(\boldsymbol{s}), \pi^2(\boldsymbol{s}), .., \pi^N(\boldsymbol{s})] \tag{11}$$

Unlike in the previous two models, π^i is not some form of argmax of another Q function but a learned function.

As in previous work, we optimize the expected reward of policy π (as predicted by a Q function) under data collected from some other policy, π_β [10, 19].

$$l_\pi = -\mathbb{E}_{\boldsymbol{s}\sim R, \boldsymbol{a}\sim\pi(\boldsymbol{s})}[Q^D(\boldsymbol{s}, \boldsymbol{a})], \tag{12}$$

where Q^D is an estimate of Q values and is trained to minimize equation 5.

We use policy gradients / REINFORCE to compute gradients of equation 12 [49]. Because π is trained off-policy, we include an importance sampling term to correct for the mismatch of the state distribution, ρ^π, from the learned policy and the state distribution, ρ^β from the behavior policy. To reduce variance, we employ a Monte Carlo baseline, G, which is the mean reward from K samples from π [23].

$$\bigtriangledown_\pi = -\mathbb{E}_{s\sim R, a\sim\pi(s)}[\frac{\rho^\pi(s)}{\rho^\beta(s)} \bigtriangledown log\pi(a|s)(Q^D(s,a) - G(s))] \tag{13}$$

$$G(s) = \frac{1}{K}\sum_k^K Q^D(s, a')|_{a'\sim\pi(s,a')} \tag{14}$$

In practice, using importance sampling ratios like this have been known to introduce high variance in gradients [28]. In this work, we make the assumption that ρ^π and ρ^β are very similar – the smaller the replay buffer R is, the better this assumption. This term can be removed and we are no longer required to compute $\beta(\boldsymbol{a}|\boldsymbol{s})$. This assumption introduces bias, but drastically lowers the variance of this gradient in practice.

During training, we approximate the highest probability path with a beam search and treat our policy as deterministic.

As is the case with off-policy algorithms, training the policy does not require samples from the environment under the target policy – only evaluations of Q^D. This makes it much more attractive for tasks where sampling the environment is challenging – such as robotics.

A figure showing the training procedure can be found in figure D.5.

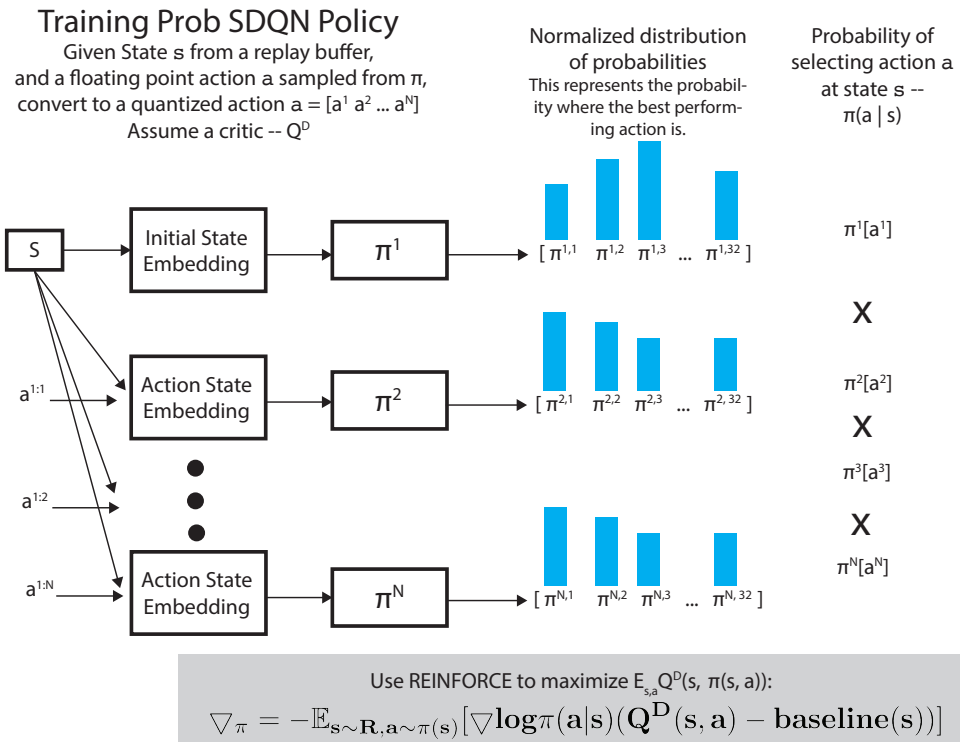

Figure D.5: Pictorial view for the Prob network showing training. Policy evaluation from this network is done in a procedure similar to that shown in Figure 2.

D.2 Network Parameterization Note

At this time, we have only tested the LSTM variant and not the untied parameterization. Optimizing these model families is ongoing and we could assume that Prob SDQN could potentially perform better if it were using the untied version as we did for the originally presented SDQN algorithm.

E Independent DQN

In the previous work, all previous methods contain both discretization and sequential prediction to enable arbitrarily complex distributions. We wished to separate these two factors, so we constructed a model that just performed discretization and keeps the independence assumption that is commonly used in RL.

Results for this method and the others presented in the appendices can be found in Appendix E.2.

E.1 Method

We define a Q function as the mean of many independent functions, F^i:

$$Q(\boldsymbol{s}, \boldsymbol{a}) = \frac{1}{N}\sum_{i=0}^{N} F^i(\boldsymbol{s}, a^i) \tag{15}$$

Because each F^i is independent of all other actions, a tractable max exists and as such we define our policy as an argmax over each independent dimension:

$$\pi_i(\boldsymbol{s}) = \underset{a^i \in \mathcal{A}^i}{\operatorname{argmax}}\, Q^i(\boldsymbol{s}, a^i) \tag{16}$$

agent	hopper	swimmer	half cheetah	humanoid	walker2d
SDQN	**3342.62**	179.23	**7774.77**	3096.71	3227.73
Prob SDQN	3056.35	**268.88**	650.33	691.11	2342.97
Add SDQN	1624.33	202.04	4051.47	**3811.44**	1517.17
IDQN	2135.47	189.52	2563.25	1032.60	668.28
DDPG	3296.49	133.52	6614.26	3055.98	**3640.93**

Table 1: Maximum reward achieved over training averaged over a 25,000 step window with evaluations ever 5,000 steps. Results are averaged over 10 randomly initialized trials with fixed hyper parameters.

As in previous models, Q is then trained with Q-learning as in equation 5.

E.2 Results

Results for the additional techniques can be seen in table 1. SDQN and DDPG are copied from the previous section for ease of reference.

The Add SDQN method performs about 800 reward better on our hardest task, humanoid, but performs worse on the simpler environments. The IDQN method, somewhat surprisingly, is able to learn reasonable policies in spite of its limited functional form. In the case of swimmer, the independent model performs slightly better than SDQN, but worse than the other versions of our models. Prob SDQN performs the best on the swimmer task by large margin, but underperforms dramatically on half cheetah and humanoid. Looking into trade offs of model design with respect to environments is of interest to us for future work.

F Training and Model details

F.1 Hyper Parameter Sensitivity

The most sensitive hyper parameters were the learning rate of the two networks, reward scale, and finally, discount factor. Parameters such as batch size, quantization amounts, and network sizes mattered to a lesser extent. We found it best to have exploration be done on less than 10% of the transitions. We didn't see any particular value below this that gave better performance. In our experiments, the structure of model used also played a large impact in performance, such as, using tied versus untied weights for each sequential prediction model.

In future work, we would like to lower the amount of hyper parameters needed in these algorithms and study the effects of various hyper parameters more thoroughly.

F.2 SDQN

In this model, we looked at a number of configurations. Of the greatest importance is the form of the model itself. We looked at an LSTM parameterization as well as an untied weight variant. The untied weight variant's parameter search is listed below.

To compute Q^i we first take the state and previous actions and do one fully connected layer of size "embedding size". We then concatenate these representations and feed them into a 2 hidden layer MLP with "hidden size" units. The output of this network is "quantization bins" wide with no activation.

Q^D uses the same embedding of state and action and then passes it though a 1 hidden layer fully connected network finally outputting a single scalar.

Hyper Parameter	**Range**	**Notes**
use batchnorm	on, off	use batchnorm on the networks
replay capacity:	2e4, 2e5, inf	
batch size	256, 512	

quantization bins	32	We found higher values generally converged to better final solutions.
hidden size	256, 512	
embedding size	128	
reward scaling	0.05, 0.1	
target network moving average	0.99, 0.99, 0.98	
adam learning rate for TD updates	1e-3, 1e-4, 1e-5	
adam learning rate for maxing network	1e-3, 1e-4, 1e-5	
gradient clipping	off, 10	
l2 reguralization	off, 1e-1, 1e-2, 1e-3, 1e-4	
learning rate decay	log linear, none	
learning rate decay delta	-2	Decay 2 orders of magnitude down from 0 to 1m steps.
td term multiplier	0.2, 0.5,	
using target network on double q network	on, off	
tree consistency multiplier	5	Scalar on the tree loss
energy use penalty	0, 0.05, 0.1, 0.2	Factor multiplied by velocity and subtracted from reward
gamma (discount factor)	0.9, 0.99, 0.999	
drag down reguralizer	0.0, 0.1	Constant factor to penalize high q values. This is used to control over exploration. It has a very small effect in final performance.
tree target greedy penalty	1.0	A penalty on MSE or Q predictions from greedy net to target. This acts to prevent taking too large steps in function space
exploration type	boltzmann or epsilon greedy	
boltzman temperature	1.0, 0.5, 0.1, 0.01, 0.001	
prob to sample from boltzman (vs take max)	1.0, 0.5, 0.2, 0.1, 0.0	
boltzman decay	decay both prob to sample and boltzman temperature to 0.001	
epsilon noise	0.5, 0.2, 0.1, 0.05, 0.01	
epsilon decay	linearly to 0.001 over the first 1M steps	

Best hyper parameters for a subset of environments.

Hyper Parameter	**Hopper**	**Cheetah**
use batchnorm	off	off
replay capacity:	inf	inf
batch size	512	512
quantization bins	32	32
hidden size	256	512
embedding size	128	128
reward scaling	0.1	0.1
target network moving average	0.99	0.9

adam learning rate for TD updates	1e-3	1e-3
adam learning rate for maxing network	5e-5	1e-4
gradient clipping	off	off
l2 regularization	1e-4	1e-4
learning rate decay for q	log linear	log linear
learning rate decay delta for q	2 orders of magnitude down from interpolated over 1M steps.	2 orders down interpolated over 1M steps
learning rate decay for tree	none	none
learning rate decay delta for tree	NA	NA
td term multiplier	0.5	0.5
useing target network on double q network	off	on
tree consistency multiplier	5	5
energy use penalty	0	0.0
gamma (discount factor)	0.995	0.99
drag down reguralizer	0.1	0.1
tree target greedy penalty	1.0	1.0
exploration type	boltzmann	boltzmann
boltzman temperature	1.0	0.1
prob to sample from boltzman (vs take max)	0.2	1.0
boltzman decay	decay both prob to sample and boltzman temperature to 0.001 over 1M steps	decay both prob to sample and boltzman temperature to 0.001 over 1M steps
epsilon noise	NA	NA
epsilon decay	NA	NA

F.3 Add SDQN

Q^D is parameterized the same as in F.2. The policy is parameterized by a multi layer LSTM. Each step of the LSTM takes in the previous action value, and outputs some number of "quantization bins." An action is selected, converted back to a one hot representation, and fed into an embedding module. The result of the embedding is then fed into the LSTM.

When predicting the first action, a learned initial state variable is fed into the LSTM as the embedding.

Hyper Parameter	**Range**	**Notes**
replay capacity:	2e4, 2e5, inf	
batch size	256, 512	
quantization bins	8, 16, 32	We found higher values generally converged to better final solutions.
lstm hidden size	128, 256, 512	
number of lstm layers	1, 2	
embedding size	128	
Adam learning rate for TD updates	1e-3, 1e-4, 1e-5	
Adam learning rate for maxing network	1e-3, 1e-4, 1e-5	
td term multiplier	1.0, 0.2, 0.5,	
target network moving average	0.99, 0.99, 0.999	
using target network on double q network	on, off	

reward scaling	0.01, 0.05, 0.1, 0.12, 0.08	
train number beams	1,2,3	number of beams used when evaluating argmax during training.
eval number beams	1,2,3	number of beams used when evaluating the argmax during evaluation.
exploration type	boltzmann or epsilon greedy	Epsilon noise injected after each action choice
boltzmann temperature	1.0, 0.5, 0.1, 0.01, 0.001	
prob to sample from boltzmann (vs take max)	1.0, 0.5, 0.2, 0.1, 0.05	
epsilon noise	0.5, 0.2, 0.1, 0.05, 0.01	

Best hyper parameters for a subset of environments.

Hyper Parameter	**Hopper**	**Cheetah**
replay capacity:	inf	inf
batch size	256	256
quantization bins	16	32
lstm hidden size	128	256
number of lstm layers	1	1
embedding size	128	
Adam learning rate for TD updates	1e-4	5e-3
Adam learning rate for maxing network	1e-5	5e-5
td term multiplier	0.2	1.0
target network moving average	0.999	0.99
using target network on double q network	on	on
reward scaling	0.05	0.12
train number beams	2	1
eval number beams	2	2
exploration type	boltzmann	
boltzmann temperature	0.1	0.1
prob to sample from boltzmann (vs take max)	0.5	0.5
epsilon noise	NA	NA

F.4 Prob SDQN

Q^D is parameterized the same as in F.2. The policy is parameterized by a multi layer LSTM. Each step of the LSTM takes in the previous action value, and outputs some number of "quantization bins". A softmax activation is applied thus normalizing this distribution. An action is selected, converted back to a one hot representation and fed into an embedding module. The result is then fed into the next time step.

When predicting the first action, a learned initial state variable is fed into the LSTM as the embedding.

Hyper Parameter	**Range**	**Notes**
replay capacity:	2e4, 2e5, inf	
batch size	256, 512	
quantization bins	8, 16, 32	We found higher values generally converged to better final solutions.
hidden size	256	
embedding size	128	

adam learning rate for TD updates	1e-3, 1e-4	
adam learning rate for maxing network	1e-4, 1e-5, 1e-6	
td term multiplier	10, 1.0, 0.5, 0.1,	
target network moving average	0.995, 0.99, 0.999, 0.98	
number of baseline samples	2, 3, 4	
train number beams	1,2,3	number of beams used when evaluating argmax during training.
eval number beams	1,2,3	number of beams used when evaluating the argmax during evaluation.
epsilon noise	0.5, 0.2, 0.1, 0.05, 0.01	
epsilon noise decay	linearly move to 0.001 over 1m steps	
reward scaling	0.0005, 0.001, 0.01, 0.015, 0.1, 1	
energy use penalty	0, 0.05, 0.1, 0.2	Factor multiplied by velocity and subtracted from reward
entropy regularizer penalty	1.0	

Best parameters from for a subset of environments.

Hyper Parameter	**Hopper**	**Cheetah**
replay capacity:	2e4	2e4
batch size	512	256
quantization bins	32	32
hidden size	256	256
embedding size	128	128
adam learning rate for TD updates	1e-4	1e-3
adam learning rate for maxing network	1e-5	1e-4
td term multiplier	10	10
target network moving average	0.98	0.99
number of baseline samples	2	4
train number beams	1	1
eval number beams	1	1
epsilon noise	0.1	0.0
epsilon noise decay	linearly move to 0.001 over 1m steps	NA
reward scaling	0.1	0.5
energy use penalty	0.05	0.0
entropy regularizer penalty	1.0	1.0

F.5 IDQN

We construct N 1 hidden layer MLP, one for each action dimension. Each mlp has "hidden size" units. We perform Bellman updates using the same strategy done in DQN [25].

We perform our initial hyper parameter search with points sampled from the following grid.

Hyper Parameter	**Range**	**Notes**
replay capacity:	inf	

batch size	256, 512	
quantization bins	8, 16, 32	
hidden size	128, 256, 512	
gamma (discount factor)	0.95, 0.99, 0.995, 0.999	
reward scaling	0.05, 0.1	
target network moving average	0.99, 0.99, 0.98	
l2 reguralization	off, 1e-1, 1e-2, 1e-3, 1e-4	
noise type	epsilon greedy	
epsilon amount(percent of time noise)	0.01, 0.05, 0.1, 0.2	
adam learning rate	1e-3, 1e-4, 1e-5	

Best hyper parameters on a subset of environments.

Hyper Parameter	**Hopper**	**Cheetah**
replay capacity:	inf	inf
batch size	512	256
quantization bins	16	8
hidden size	512	128
gamma (discount factor)	0.99	0.995
reward scaling	0.1	0.1
target network moving average	0.99	0.99
l2 reguralization	off	1e-4
noise type	epsilon greedy	epsilon greedy
epsilon amount(percent of time noise)	0.05	0.01
adam learning rate	1e-4	1e-4

F.6 DDPG

Due to our DDPG implementation, we chose to do a mix of random search and parameter selection on a grid.

Hyper Parameter	**Range**	**Notes**
learning rate :	[1e-5, 1e-3]	Done on log scale
gamma (discount factor)	0.95, 0.99, 0.995, 0.999	
batch size	[10, 500]	
actor hidden 1 layer units	[10, 300]	
actor hidden 2 layer units	[5, 200]	
critic hidden 1 layer units	[10, 400]	
critic hidden 2 layer units	[4, 300]	
reward scaling	0.0005, 0.001, 0.01, 0.015, 0.1, 1	
target network update rate	[10, 500]	
target network update fraction	[1e-3, 1e-1]	Done on log scale
gradient clipping from critic to target	[0, 10]	
OU noise damping	[0, 1]	
OU noise std	[0, 1]	

Best hyper parameters on a subset of environments.

Hyper Parameter	**Hopper**	**Cheetah**
learning rate :	0.00026	0.000086
gamma (discount factor)	0.995	.995
batch size	451	117
actor hidden 1 layer units	48	11
actor hidden 2 layer units	107	199
critic hidden 1 layer units	349	164
critic hidden 2 layer units	299	256
reward scaling	0.01	0.01
target network update rate	10	445
target network update fraction	0.0103	0.0677
gradient clipping from critic to target	8.49	0.600
OU noise damping	0.0367	0.6045
OU noise std	0.074	0.255