STOCHASTIC APPROXIMATION: THEORY AND APPLICATIONS

Dr. Gugan Thope

Department of Computer Science and Engineering

Indian Institute of Science, Bangalore

Week 13

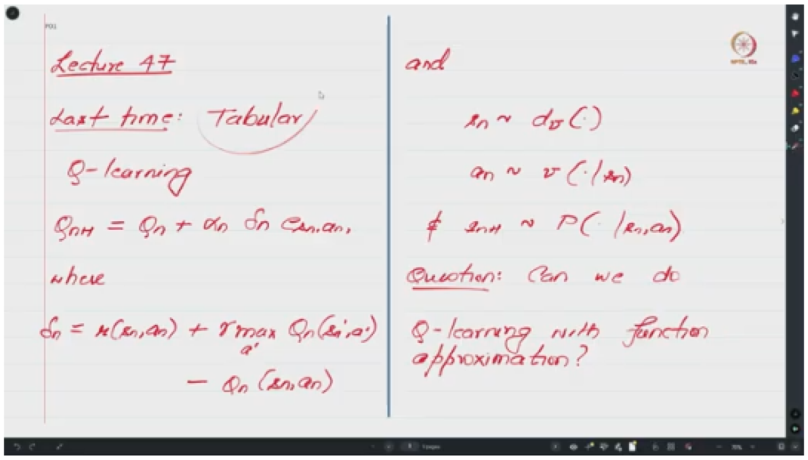
Lecture 47

Q-Learning with Linear Function Approximation — A Unified Switching Systems Perspective

Hello and Namaste everyone, welcome to lecture 47 of this course on Stochastic Approximation. So, in the last two weeks we looked at applications of reinforcement learning, we will be continuing this discussion over the next couple of lectures as well. And if you remember in the previous week, we sort of looked at the analysis of tabular Q learning. That is we wanted to identify the optimal policy and in wanting to do so, we focused on identifying or estimating the Q value function associated with the optimal policy which we denoted it as Q star. And as we said Q star lives in this SA dimensional space and in general it will be very large but suppose we are in settings where this SA dimension is not very large and we can indeed work with these SA dimensional vectors then we came up with an algorithm which we refer to as the Q learning algorithm to get to estimate this Q star directly.

And once we estimate this Q star, the greedy policy associated with Q star we showed was optimal. And, you know, we discussed the convergence of this algorithm and so on and so forth. And in today's class, we are going to take this discussion further. In particular, we are going to ask, you know, what happens if we try to do Q learning in a large state action setup, in which case we are forced to work with function approximation. And the idea then or the goal then would be to ask would Q learning with a function approximation version still help us or not.

So that is the question that we are going to ask in the next couple of lectures and let us see what is the answer that we get to that. So let us begin our formal discussion. So as I said last time we looked at tabular Q learning or the version of Q learning with no function approximation. So whenever I say tabular I mean it is without function approximation and the update rule that we looked at had the following form that is Qn plus 1 is Qn plus alpha n times some delta n times E of SNAN.



Lecture 47

Last time: Tabular

Q-learning

$Q_{n+1} = Q_n + \alpha_n \delta_n e_{s_n, a_n}$,

where

$\delta_n = r(s_n, a_n) + \gamma \max_{a'} Q_n(s_n', a') - Q_n(s_n, a_n)$

and

$s_n \sim d_\nu(\cdot)$

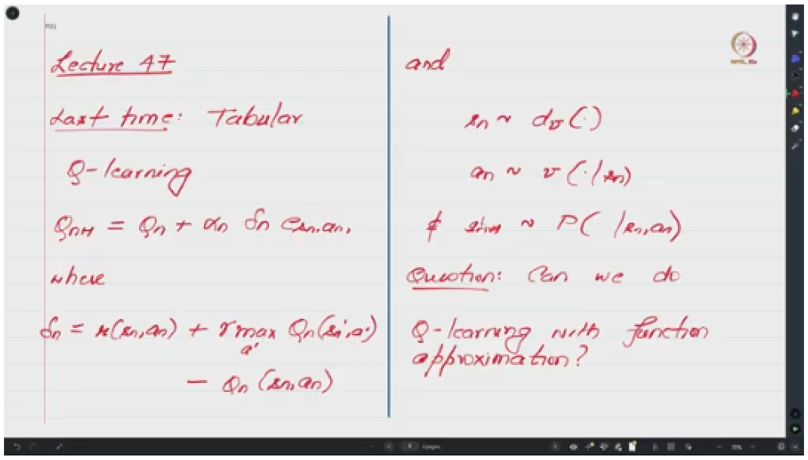
$a_n \sim \nu(\cdot | s_n)$

& $s_{n+1} \sim P(\cdot | s_n, a_n)$

Question: Can we do Q-learning with function approximation?

This E SNAN or E SA in general is a standard basis vector sitting in the R cardinality of S times cardinality of A Euclidean space and E SA is the vector whose SAth coordinate that is little s comma little ath coordinate is 1 and all other entries are 0. So using that we can define this ESNAN vector and delta N is a scalar which is known as the TD error. And the TD error basically looks at the difference between a new estimate of Q star SA and subtracts it or looks at the difference between a new estimate of Q star SA and the previous estimate of Q star SA. So the previous estimate of Q star SNAN is this and this is the new estimate of Q star SNAN and we look at the difference between the two and that is why this is referred to as the temporal difference of your Q star estimate and then we use this temporal difference in the following way in the algorithm and we showed that this algorithm converges to Q star which is the Q value associated with the optimal policy.

And we had emphasized that these essence are chosen from the stationary distribution associated with some behavior policy. The word behavior policy means that we interact with the environment using some policy and that policy could be very different from your optimal policy. So the behavior policy is one with which we interact with the environment and that could be very different and we have forced to do that because we do not know the optimal policy. And this d nu over here is the stationary distribution associated with the behavior policy or the Markov chain induced by the behavior policy and this action a n is chosen according to your behavior policy while s n prime. So I should perhaps not put s n plus 1 here but s n prime over here.



Lecture 47

Last time: Tabular Q-learning

$Q_{n+1} = Q_n + \alpha_n \delta_n e_{s_n, a_n}$,

where

$\delta_n = r(s_n, a_n) + \gamma \max_{a'} Q_n(s_n', a') - Q_n(s_n, a_n)$

and

$s_n \sim d_\nu(\cdot)$

$a_n \sim \nu(\cdot / s_n)$

$\&$ $s_{n+1} \sim P(\cdot / s_n, a_n)$

Question: Can we do Q-learning with function approximation?

So s n prime is chosen or sampled from your transition kernel. Is this okay? And we said that, you know, if you do it in this way and if nu is arbitrary, right, and as long as these quantities are strictly positive and so on and so forth, one can show that this algorithm that we have over here indeed converges to Q star.

$$Q_{n+1} = Q_n + \left(\alpha_n \delta_n e s_n, a_n\right)$$

$$\delta_n = r\left(s_n, a_n\right) + rQ_n\left(s_n^{'}, a^{'}\right) - Q_n\left(s_n, a_n\right)$$

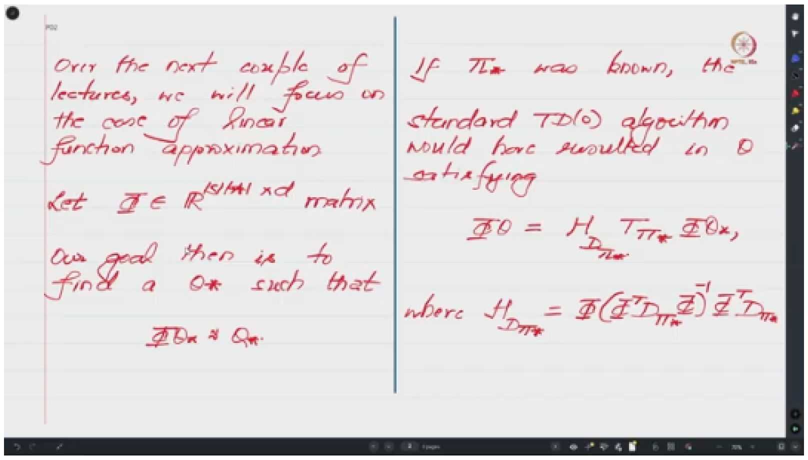
$$s_n \sim d_\nu(\cdot)$$

$$a_n \sim \nu\left(s_n\right)$$

$$s_{n+1} \sim P\left(s_n, a_n\right)$$

So, the question that we would like to discuss over the next couple of lectures is can we do Q learning with function approximation. So, as I told you when we are in a setting where the states and action spaces are very very large we are forced to work with function approximation.

So, the question is: can we somehow use some variant of this algorithm when we are forced to work with function approximation? In particular, in this class and the next couple of lectures, we will focus on, in some sense, the relatively easy case of linear function approximation. In which we have been given some feature matrix phi. So, phi is a matrix of size—or the number of rows equal to the product of the cardinality of the state and action spaces—and the number of columns equals D. Typically, we will presume that this value D is sufficiently small as compared to the product SA. So, instead of searching in the whole of RSA space, what we will do is search in the column space of this matrix phi.



Over the next couple of lectures, we will focus on the case of linear function approximation

Let $\Phi \in \mathbb{R}^{|S||A| \times d}$ matrix

Our goal then is to find a θ_* such that

$$\Phi\theta_* \approx Q_*.$$

If π_* was known, the standard TD(0) algorithm would have resulted in θ satisfying

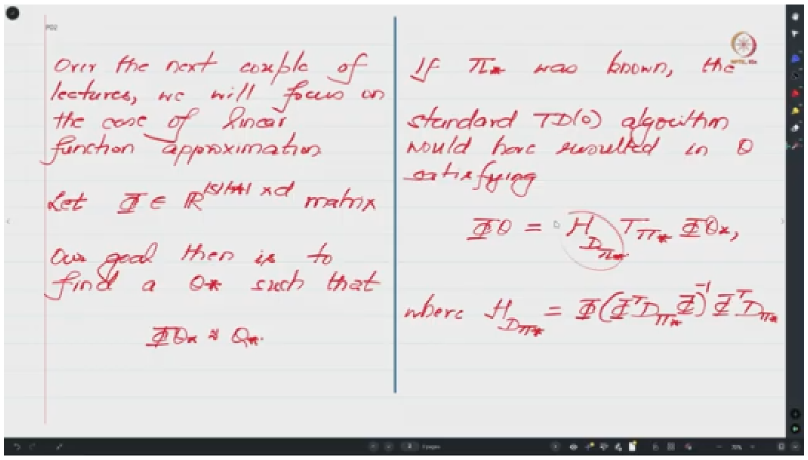
$$\Phi\theta = \Pi_{D_{\pi_*}} T_{\pi_*} \Phi\theta_*,$$

where $\Pi_{D_{\pi_*}} = \Phi(\Phi^T D_{\pi_*} \Phi)^{-1} \Phi^T D_{\pi_*}$

Where the column space is d-dimensional, and hence we are restricting our search space to a smaller space. The hope is that by restricting ourselves to a smaller space, we will reduce the computational effort and time, right? And we are able to find a vector theta star such that phi theta star is approximately equal to q star. So, phi theta star approximates q star very well. Then, by looking at the greedy policy that is associated with phi theta star, we would have a good estimate of the optimal policy itself.

So, this is the goal. In other words, we want to find an estimate of Q star within the column space of phi, and this restricted search is what we will refer to as trying to do Q-learning with function approximation. So, of course, if pi star was known—so recall that q star is the q-value associated with the optimal policy, which we are denoting by pi star. So, if pi star was known, two weeks ago, we had discussed the problem of policy evaluation. So, if pi star is known, finding q star is equivalent to evaluating the value—or the q-value, in particular—of this policy pi star.

Is this okay? And in that case, that is when pi star would have been known, we could have used the standard TD0 algorithm to get an estimate of pi star, right? In particular, the TD0 algorithm could have been used to find the fixed point of the projected Bellman operator. So, what is the projected Bellman operator? It is the product of this matrix or projection matrix H subscript d pi star, I will tell you what this means, times d pi star.



Over the next couple of lectures, we will focus on the case of linear function approximation

Let $\Phi \in \mathbb{R}^{|S||A| \times d}$ matrix

Our goal then is to find a θ_* such that

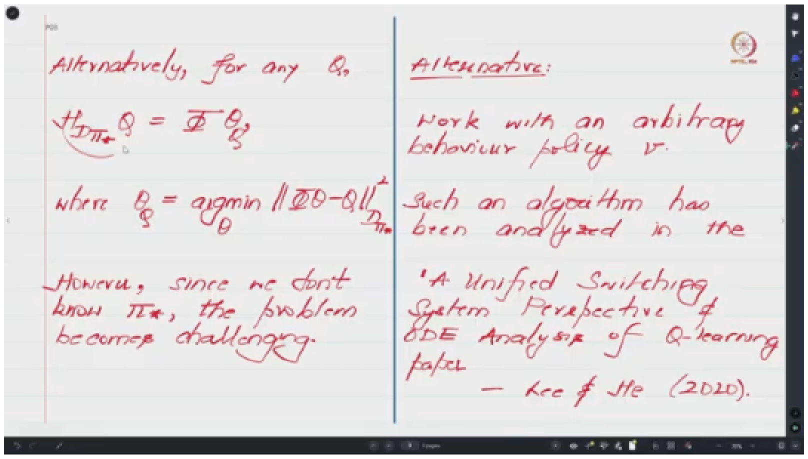
$$\Phi\theta_* \approx Q_*.$$

If π_* was known, the standard TD(0) algorithm would have resulted in θ satisfying

$$\Phi\theta = H_{D_{\pi_*}} T_{\pi_*} \Phi\theta_*,$$

where $H_{D_{\pi_*}} = \Phi(\Phi^T D_{\pi_*} \Phi)^{-1} \Phi^T D_{\pi_*}$

So, if pi star was known, T pi star is your Bellman operator associated with policy pi star, and this H d pi star is the matrix that is given over here. One can see that this H d pi star basically does the job of projection. In other words, if I am given some arbitrary vector Q and multiply it with H d pi star, then the output that I will get can be represented as some phi theta Q, where theta Q is basically the theta vector that minimizes the distance mentioned over here. Notice that the distance is influenced by d pi star, where d pi star is the S A cross S A matrix. In which, on the diagonal entries, you see the stationary distribution induced by this pi star policy, and it is multiplied by pi star of A given S. In other words, this d pi star. So, the s-a-th diagonal entry of d pi star would be d little d pi star of s times pi star a given s. So, this is slightly different from what we had in the TD0 algorithm that we had



Alternatively, for any Q,

$\Pi_{D_{\pi_*}} Q = \Phi \theta_Q$,

where $\theta_Q = \underset{\theta}{\arg\min} \|\Phi\theta - Q\|^2_{D_{\pi_*}}$

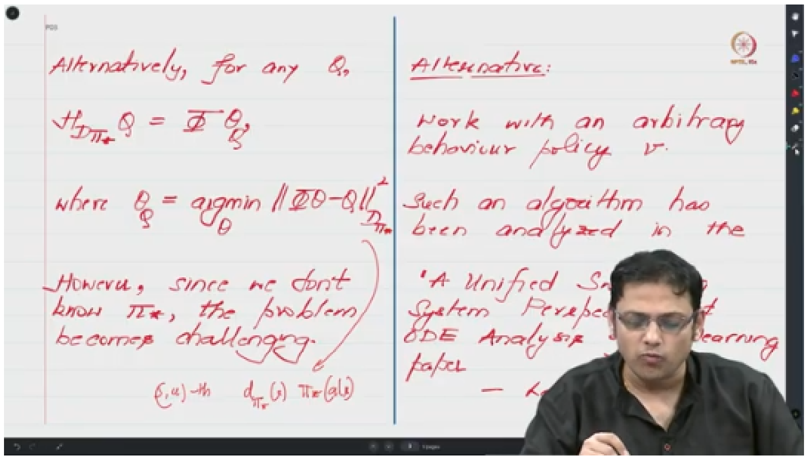
However, since we don't know π_*, the problem becomes challenging.

Alternative:

Work with an arbitrary behaviour policy ν.

Such an algorithm has been analyzed in the 'A Unified Switching System Perspective of ODE Analysis of Q-learning' paper

— Lee & He (2020).



Alternatively, for any Q,

$\Pi_{D_{\pi_*}} Q = \Phi \theta_Q$,

where $\theta_Q = \underset{\theta}{\arg\min} \|\Phi\theta - Q\|^2_{D_{\pi_*}}$

However, since we don't know π_*, the problem becomes challenging.

Alternative:

Work with an arbitrary behaviour policy ν.

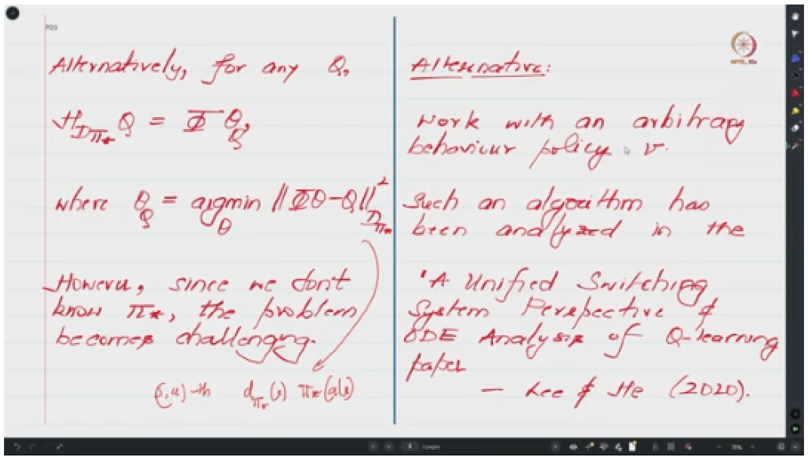
Such an algorithm has been analyzed in the

(s,a)-th $\quad d_{\pi_*}(s)\,\pi_*(a|s)$

seen two weeks back. There, this diagonal matrix was S cross S; however, since we are working with Q value functions, these matrices will now be SA cross SA, where S by S I mean the cardinality of the state space and A I mean the cardinality of the action space, right. So, as I told you, if we had known pi star, then the TD0 algorithm would have found the projection of—I mean, I should say that the TD0 algorithm would have found the fixed point of the projected Bellman operator. Is this okay? So, this is what we have.

We would have got if we had run the TD0 algorithm. However, the challenge is that we do not know pi star. If we had known pi star it would have finding Q pi star would have reduced to the problem of policy evaluation. However, we do not know pi star hence the problem becomes challenging. So now the question is you know with which policy should we interact with the environment.

So one possibility is that we can pick some arbitrary behavior policy mu like we did in the tabular Q learning case right and then you know hope that whatever algorithm that we try to come up with right that algorithm manages to find a good enough estimate of Q star. And such an algorithm has indeed been analyzed in this paper called a unified switching system perspective and OD analysis of Q learning by Dongguan Li and Niao He from 2020. So you can look at the details of such an algorithms analysis. from this paper and I would like to highlight that whatever tools that we studied in the previous two weeks those tools can indeed be used to you know study you know the algorithm where the states and actions are sampled by using some fixed behavior policy. Is this okay?



Alternatively, for any Q,

$\mathcal{H}_{D_{\pi_*}} Q = \Phi \theta_Q$,

where $\theta_Q = \underset{\theta}{\operatorname{argmin}} \|\Phi\theta - Q\|^2_{D_{\pi_*}}$

However, since we don't know π_*, the problem becomes challenging.

(s,a)-th $\quad d_{\pi_*}(s)\, \pi_*(a|s)$

Alternative:

Work with an arbitrary behaviour policy ν.

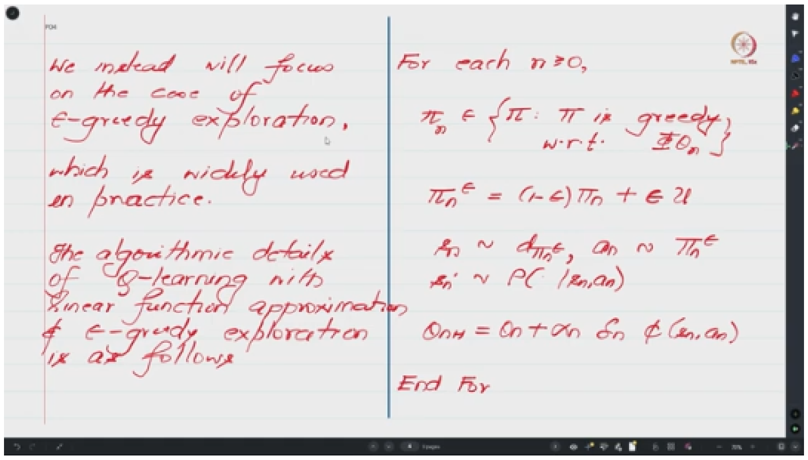
Such an algorithm has been analyzed in the

'A Unified Switching System Perspective of ODE Analysis of Q-learning paper

— Lee & He (2020).

Which could be very different from pi star. However, if you look at that paper, you would notice that you first of all require some strong conditions under which we guarantee convergence. And even if we guarantee convergence, the limit to which the algorithm converges to depends on the choice of your behavior policy. So, there is no guarantee that you know such an algorithm if it converges that limit would actually be correlated with the optimal policy. Instead the limit will actually depend on this choice of this behavior policy and hence the choice of the behavior policy becomes very crucial.

So, keeping that in mind, we will not be looking at the analysis of the algorithm where the behavior policy is new. Instead, we will focus on the case of what is referred to as epsilon-greedy exploration. On the one hand, it is quite challenging to analyze. On the other hand, the reason why we want to focus on this is because this is one of the algorithms that is widely used in practice. Now, the algorithmic details of Q-learning with linear function approximation and epsilon-greedy exploration are as follows.



We instead will focus on the case of ϵ-greedy exploration, which is widely used in practice.

The algorithmic details of Q-learning with linear function approximation & ϵ-greedy exploration is as follows

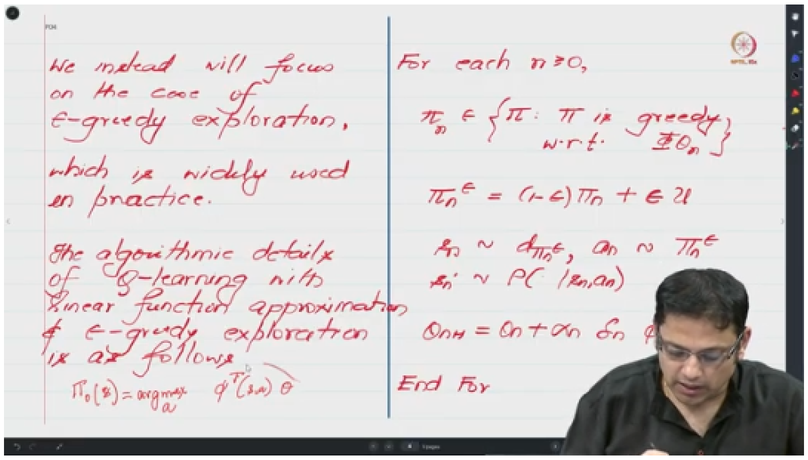
```
For each $n \geq 0$,
    $\pi_n \in \{\pi : \pi \text{ is greedy w.r.t. } \Phi\theta_n\}$
    $\pi_n^\epsilon = (1-\epsilon)\pi_n + \epsilon\,\mathcal{U}$
    $s_n \sim d_{\pi_n^\epsilon}, \; a_n \sim \pi_n^\epsilon$
    $s_n' \sim P(\cdot \mid s_n, a_n)$
    $\theta_{n+1} = \theta_n + \alpha_n \delta_n \phi(s_n, a_n)$
End For
```

So, here is the algorithm associated with Q-learning with linear function approximation and epsilon-greedy exploration. I am going to give an idealized version of that algorithm so that the analysis becomes slightly less challenging. But in practice, one works with a more sophisticated version of this algorithm whose details I will let you know as we move on. Is this okay? So, what are the details of this algorithm?

So, what we do is we will begin with some theta 0. So, theta 0 and, in particular, phi theta 0 can be viewed as an initial estimate of Q star. So, we will somehow start with this. Then, at the 0th iteration, what we will do is we will first pick a policy pi 0 which is greedy with respect to phi theta 0. So, by greedy, I mean phi theta 0 will be a vector of size cardinality SA.

So let me just write it. So phi theta 0 will be a vector of size cardinality S times cardinality A. What we will do is we will presume that pi 0 of at state S, right? So, the action that this policy picks at state S is basically argmax over A, okay, of phi transpose SA comma theta, right?

So, whatever is this S over here, you take that S and you take this inner product for different values of A, right? And maybe I should highlight that this is theta 0. So, you take this inner product for different values of A and take that action which maximizes this inner product. If there are multiple actions, we will break ties arbitrarily. So we will just pick one action which maximizes this inner product over here, and that is the action that will be taken by this pi 0 policy.



We instead will focus on the case of ϵ-greedy exploration, which is widely used in practice.

The algorithmic details of Q-learning with linear function approximation & ϵ-greedy exploration is as follows

$\pi_0(s) = \arg\max_a \phi^T(s,a)\,\theta$

For each $n \geq 0$,

$\pi_n \in \{\pi : \pi \text{ is greedy w.r.t. } \Phi\theta_n\}$

$\pi_n^\epsilon = (1-\epsilon)\pi_n + \epsilon\,\mathcal{U}$

$s_n \sim d_{\pi_n^\epsilon}, \quad a_n \sim \pi_n^\epsilon$

$s_n' \sim P(\cdot \mid s_n, a_n)$

$\theta_{n+1} = \theta_n + \alpha_n \delta_n \phi$

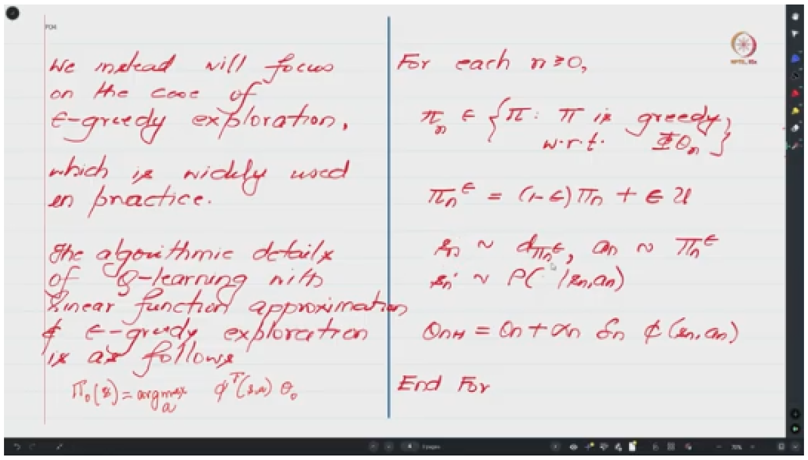
End For

So in other words, this pi 0 policy that is for n equals 0, this pi 0 will be chosen to be greedy with respect to phi theta 0, where greedy is defined in this way. Then what we will do is we will use this pi 0 to cook up an epsilon-greedy version of this policy pi 0. So what does epsilon-greedy mean? It is specified over here. With probability 1 minus epsilon, we will act according to your policy pi 0, and with probability epsilon, we will pick an action uniformly randomly.

So this is like a uniformly random policy. So the idea here is that You know, if your phi theta n, right, so if you are in a general iteration n, if phi theta n is a good estimate of q star, then your pi n will behave like your optimal policy. However, if your phi theta n is a very poor estimate of, you know, pi star, then with some probability, you are not acting according to pi star, and that sort of allows you to explore a bit more. The hope is that by doing this exploration with probability epsilon, which is different from the greedy policy, the hope is that whenever we get stuck at some bad policy, we would be able to escape from that policy.

So that is what this epsilon's role over here is, and because of this choice, we refer to this algorithm—I mean, we say that this algorithm actually uses epsilon-greedy exploration. Okay, so let us just quickly summarize what we have done so far. At iteration 0, we have phi theta 0, and we, you know, pick pi 0, which is greedy with respect to phi theta 0. Then we construct the epsilon-greedy policy associated with pi 0. So in this epsilon-greedy policy,

with probability epsilon, you pick a uniform action, and with probability 1 minus epsilon, you act according to pi n. So what does that mean? What you do is you first pick a state, which is sampled from the stationary distribution of the Markov chain induced by pi n epsilon. So that is the idea. How do we get this?



We instead will focus on the case of ϵ-greedy exploration, which is widely used in practice.

The algorithmic details of Q-learning with linear function approximation & ϵ-greedy exploration is as follows

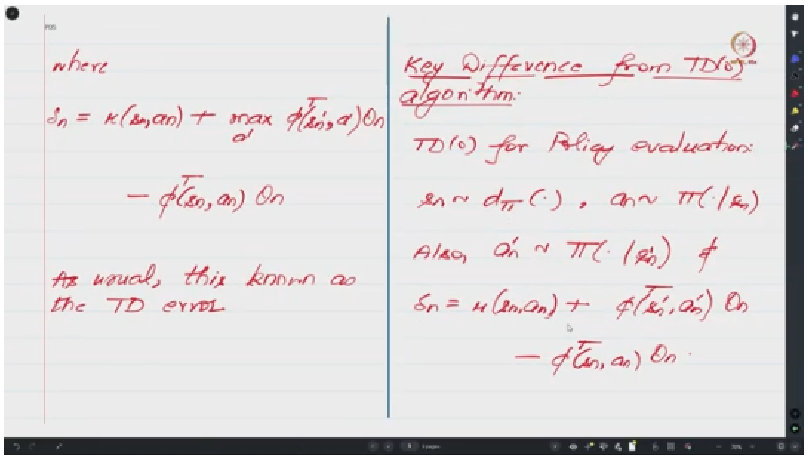
$\pi_0(s) = \arg\max_a \phi^T(s,a)\,\theta_0$

```
For each $n \geq 0$,
    $\pi_n \in \{\pi : \pi \text{ is greedy w.r.t. } \Phi\theta_n\}$
    $\pi_n^\epsilon = (1-\epsilon)\pi_n + \epsilon\,\mathcal{U}$
    $s_n \sim d_{\pi_n^\epsilon}$, $a_n \sim \pi_n^\epsilon$
    $s_n' \sim P(\cdot \mid s_n, a_n)$
    $\theta_{n+1} = \theta_n + \alpha_n \delta_n \phi(s_n, a_n)$
End For
```

Again, as I told you, you know, if your Markov chain is ergodic and if you allow the Markov chain to evolve for some time starting from some arbitrary state, one can, you know, pick the state that appears after a sufficiently large number of iterations, and one can then see that the terminal state can actually be presumed to be sampled from this distribution over here. So, you sample Sn in this fashion and An from this Pi n epsilon. So, whatever your state is, you pick the action as per your epsilon-greedy policy. And once you have this state and this action, you sample your next state, that is Sn prime, from your transition kernel.

So, in this way, you would have Sn, An, and Sn prime. And using these three quantities, you define your update rule in the following fashion. So, the update rule over here is given by theta n plus 1 equals theta n plus alpha n times the temporal difference by phi of Sn An. So, one can ask, how did we come up with this algorithm? Well, we again go back to our strategy of designing the TD0 algorithm.

So, if you remember, in the TD0 algorithm, we also had an update rule which was very similar to this. So, whatever recipe we followed there, if we follow the same recipe, one can see that we will come up with an update rule of the following form. And the difference between that update rule and this update rule, I will soon talk about. But firstly, let me elaborate on what delta n over here is. So, delta n again is referred to as the temporal difference or the TD error, and it is given in the following way.



where

$$\delta_n = R(s_n,a_n) + \max_{a'} \phi^T(s_n', a')\theta_n - \phi^T(s_n,a_n)\theta_n$$

As usual, this known as the TD error

Key Difference from TD(0) algorithm:

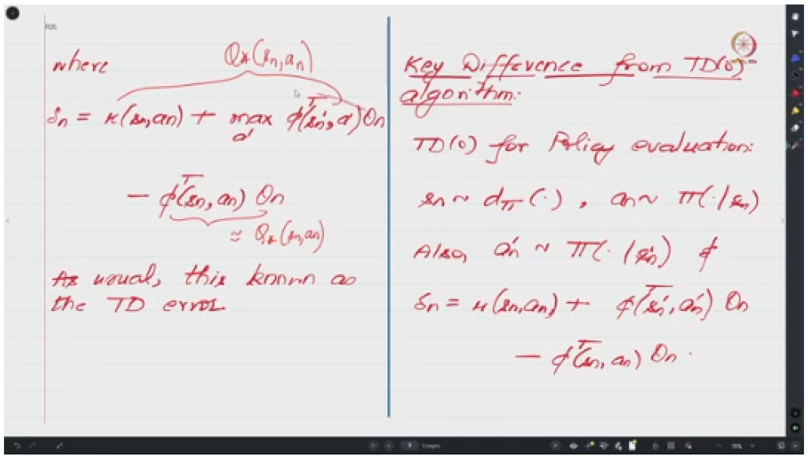
TD(0) for Policy evaluation:

$s_n \sim d_\pi(\cdot)$, $a_n \sim \pi(\cdot/s_n)$

Also, $a_n' \sim \pi(\cdot/s_n')$ &

$$\delta_n = R(s_n,a_n) + \phi^T(s_n',a_n')\theta_n - \phi^T(s_n,a_n)\theta_n .$$

So, delta n is R Sn An plus max over A prime phi Sn prime A prime transpose theta n. This inner product you take for different A primes and then take the max over A prime over here and then subtract it with phi transpose Sn An theta n. So, you can think of this quantity over here as approximately, you know, or the, you know, estimate of Q star S N A N at time instance N, and this can be viewed as another estimate of Q star. So, this is based on, you know—so let me just do it carefully here so that there is no confusion. So, this should be SnAn. So, this is one estimate of Q star SnAn, and this is another estimate of Q star SnAn, and hence this can be viewed as the temporal difference or the TD error.



where

$$\delta_n = \overbrace{r(s_n,a_n) + \max_{a'} \phi^T(s_n',a')\theta_n}^{Q_*(s_n,a_n)} - \underbrace{\phi^T(s_n,a_n)\theta_n}_{\simeq Q_*(s_n,a_n)}$$

~~As~~ usual, this known as the TD error

Key Difference from TD(0) algorithm:

TD(0) for Policy evaluation:

$s_n \sim d_\pi(\cdot)$, $a_n \sim \pi(\cdot/s_n)$

Also, $a_n' \sim \pi(\cdot/s_n')$ &

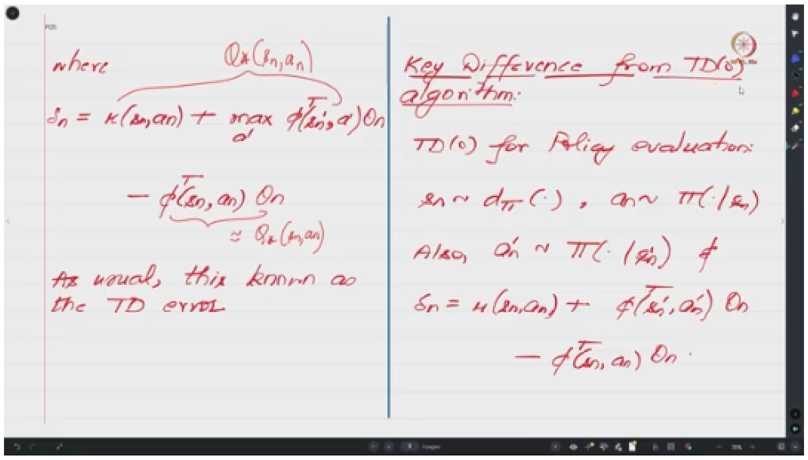
$$\delta_n = r(s_n,a_n) + \phi^T(s_n',a_n')\theta_n - \phi^T(s_n,a_n)\theta_n.$$

So, this is the way we are going to run our Q-learning algorithm. So, let me go back and summarize. So, at any time instance n, we pick a greedy policy that is associated with respect to phi theta n. Using the greedy policy, we construct the epsilon-greedy policy. Then we sample the current state using the stationary distribution of your epsilon-greedy policy, pick the action according to the epsilon-greedy policy, and get the next state, that is Sn prime, from the transition kernel. And then we run our usual, you know, TD update algorithm, right?

And now one can ask, what is the key difference between the TD0 algorithm for policy evaluation and this Q-learning algorithm with linear function approximation that we have stated? The key difference is how do we sample SN and AN, right? So, you know, in the Q-learning algorithm, this SN and AN, right, especially with epsilon-greedy exploration, this SN was sampled from your stationary distribution associated with this epsilon-greedy policy, and AN similarly is chosen according to your epsilon-greedy policy. On the other hand, when we want to do policy evaluation—so first of all, notice that in policy evaluation, we have been given some policy pi, some fixed policy pi, and our goal is to evaluate that policy. By evaluate, I mean we want to know what is the value of Q pi.

I mean, in the previous case, we had focused on estimating V pi, but here we are going to focus on an analogous problem which involves estimating Q pi. That is the Q-value function associated with this policy pi. So in this case, your current state Sn is supposed to be sampled from the stationary distribution that is associated with your behavior policy or the Markov chain induced by your policy Pi itself. So the policy that we want to evaluate is the behavior policy or the policy with which we will interact with the environment. So in the policy evaluation case, this makes sense because somebody has given us a policy or a strategy, and they are asking us how good that strategy is.

So that is the policy evaluation problem, and what that means is that we know this policy pi, and since we know this policy pi, we can interact with the environment with policy pi itself, and hence this statement makes sense. In other words, Sn is sampled from the stationary distribution associated with your behavior policy Pi, and An is sampled again from this policy Pi whose value you wish to estimate. Similarly, this An A prime that is over here will be replaced by An prime, and we won't have any max operation. In other words, the TD error in the case of the TD0 algorithm would have An prime without the max. So, this An prime again is presumed to be sampled from your policy pi itself, which is what we wish to evaluate.



where

$$\delta_n = r(s_n, a_n) + \overbrace{\max_{a'} \phi^T(s_n', a')\theta_n}^{Q_*(s_n', a_n')} - \underbrace{\phi^T(s_n, a_n)\theta_n}_{\simeq Q_*(s_n, a_n)}$$

As usual, this known as the TD error

Key Difference from TD(0) algorithm:

TD(0) for Policy evaluation:

$s_n \sim d_\pi(\cdot)$, $a_n \sim \pi(\cdot / s_n)$

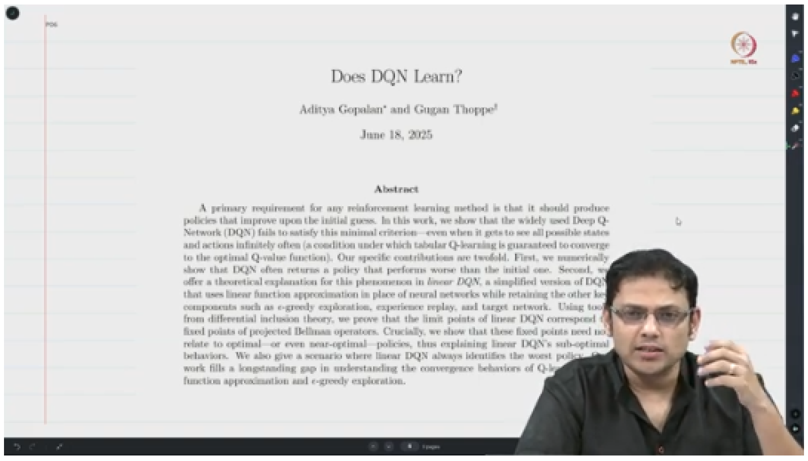
Also, $a_n' \sim \pi(\cdot / s_n')$ &

$$\delta_n = r(s_n, a_n) + \phi^T(s_n', a_n')\theta_n - \phi^T(s_n, a_n)\theta_n.$$

So, in this case, while the update rule for the TD0 algorithm and the Q-learning algorithm with linear function approximation in both cases may look similar, the way we interact with the environment or the way we collect the experience trajectory differs. In the Q-learning case, especially Q-learning with epsilon-greedy exploration, we collect these experiences or these Sn, An, Sn prime tuples using your evolving behavior policy. And that behavior policy is greedy with respect to your Q-star estimate. On the other hand, in the policy evaluation case, The states and actions are sampled according to the behavior policy itself.

I mean, I should be careful about the policy whose value we are to estimate. So that policy itself will be treated as the behavior policy, and we will interact with the environment using that policy itself. So, other than these differences, the update rules may look very similar, but because of these small differences, the purpose and the behavior of the algorithms will significantly differ. So, two weeks back, we analyzed the policy evaluation algorithm. Today, while the update rule may look very similar to that, we are analyzing the Q-learning algorithm.

So, in a recent paper of ours, we actually looked at a similar algorithm like this, which is known as Q-learning with linear function approximation, and we asked what we can say about the limiting behavior of this algorithm. So, those who wish to get additional details about this work can look up this paper of ours. This is joint work with my colleague Professor Aditya Gopalan from ISC. And here, what we did was—before we begin the analysis—first, let us show you some interesting simulation results that we observed. So, what we did was we looked at a two-state, two-action MDP, right? And so, in this case, we



Does DQN Learn?

Aditya Gopalan* and Gugan Thoppe†

June 18, 2025

Abstract

A primary requirement for any reinforcement learning method is that it should produce policies that improve upon the initial guess. In this work, we show that the widely used Deep Q-Network (DQN) fails to satisfy this minimal criterion—even when it gets to see all possible states and actions infinitely often (a condition under which tabular Q-learning is guaranteed to converge to the optimal Q-value function). Our specific contributions are twofold. First, we numerically show that DQN often returns a policy that performs worse than the initial one. Second, w offer a theoretical explanation for this phenomenon in *linear DQN*, a simplified version of DQ that uses linear function approximation in place of neural networks while retaining the other ke components such as ϵ-greedy exploration, experience replay, and target network. Using too from differential inclusion theory, we prove that the limit points of linear DQN correspond fixed points of projected Bellman operators. Crucially, we show that these fixed points need no relate to optimal—or even near-optimal—policies, thus explaining linear DQN's sub-optimal behaviors. We also give a scenario where linear DQN always identifies the worst polic work fills a longstanding gap in understanding the convergence behaviors of Q-le function approximation and ϵ-greedy exploration.

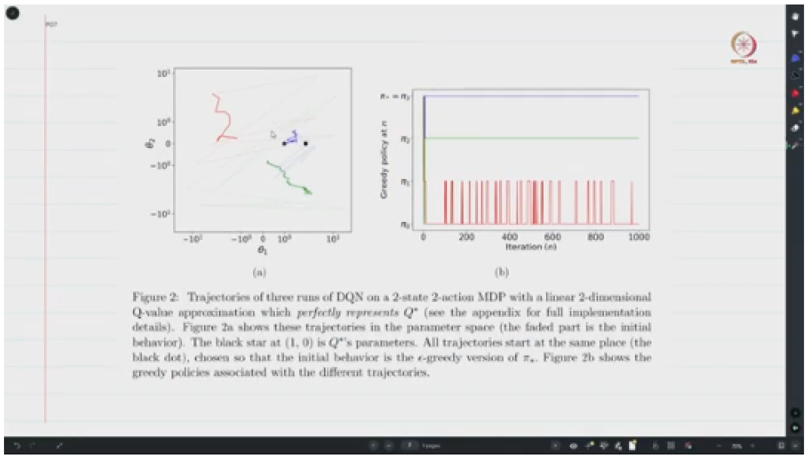


Figure 2: Trajectories of three runs of DQN on a 2-state 2-action MDP with a linear 2-dimensional Q-value approximation which *perfectly represents* Q^* (see the appendix for full implementation details). Figure 2a shows these trajectories in the parameter space (the faded part is the initial behavior). The black star at (1, 0) is Q^*'s parameters. All trajectories start at the same place (the black dot), chosen so that the initial behavior is the ϵ-greedy version of π_*. Figure 2b shows the greedy policies associated with the different trajectories.

The cardinality of the state space is 2, and the cardinality of the action space is 2. So, their product is 4. So, your q, pi's, and q star's, they live in a 4-dimensional space. And what we did was we picked a phi whose number of columns was 2. So, in this case, we work with a linear function approximation where the column space of phi is 2-dimensional.

And in this setting, What we additionally did was we presumed or ensured that the column space of phi has Q star. In other words, we ensured that one of the columns of your phi matrix actually includes Q star. So, the linear function approximation class that we are working with is perfectly realizable because it includes Q star. So, in other words, if we try to find a good estimate of Q star within the column space of phi that we are using here, ideally we would like to reach that Q star representation itself because Q star lies in the column space of this phi.

So what we did was we ran this Q-learning algorithm that I described in the previous slide, right? And here are some three trajectories that we saw, right? So this black dot over here is basically the point from which all three trajectories start. First, let me maybe tell you what this plot over here is.

So as I told you, we used a two-dimensional linear function approximation. So any vector in this space can be represented using two coordinates. So those two coordinates are theta1 and theta2. In other words, phi theta1 will be the vector in the column space of phi associated with the vector theta1, theta2. So, theta1, theta2 basically characterizes the choice of theta.

And you know this black dot over here specifies theta 0. So this theta 0 or phi times theta 0 is the initial estimate of Q star. And this is where we start our Q-learning with linear function approximation algorithms. And if you zoom in closely, you can see that there are three trajectories with three colors. One is in blue, one is in green, and one is in red.

And the star over here corresponds to the theta star value, which is associated with the Q value of your optimal policy. And you can see that it sits at 1, 0. As I told you, we had set one of the columns of your phi matrix to be Q star itself. In particular, the first column of this phi matrix was Q star, and hence this 1, 0 value corresponds to the theta star such that phi theta star is Q star.

So this is where your theta star lies. This is theta 0, and we ran multiple trajectories; here are the behaviors of three representative trajectories. You can see that for every color, there is some faded part and some solid part. The faded part basically corresponds to the initial behavior of the trajectory, and the solid part corresponds to the tail behavior. Is this okay?

And one can see that, you know, you have a lot of these zigzag behaviors. This is because there is noise in your update rules, and the noise will sort of make you jump around, but eventually, as the step size becomes smaller and smaller, okay, your iterate starts stabilizing and will go somewhere. And one can see that for the red trajectory, it seems to be converging over here. The green trajectory seems to be converging over here, and the blue trajectory alone seems to be converging to this star, which corresponds to the theta star associated with your optimal Q value function. Now, this behavior is slightly surprising—at least it was very surprising to us—because the initial value, that is theta 0, that we have chosen is very close to theta star.

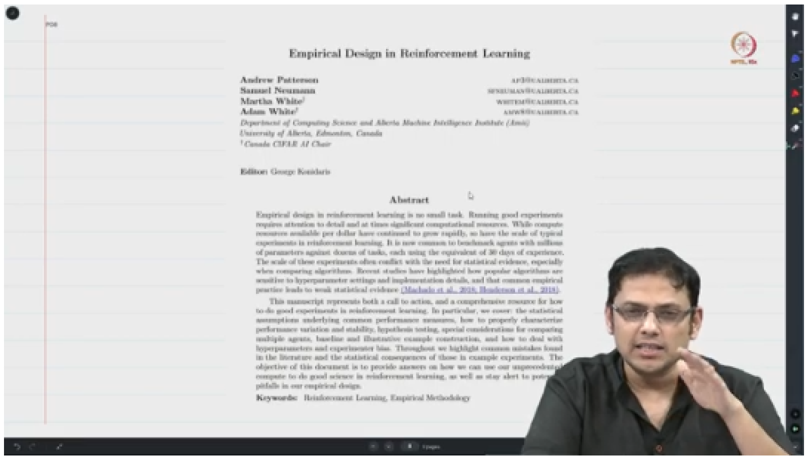
But despite this, you see that the algorithm that is initiated at phi theta 0 actually tries to reach multiple places; in particular, in some cases, it tries to settle here, in some plays, in some runs of the algorithm, it tries to settle over here, and only on a small subset of the runs does it actually manage to find Q star. So this behavior sort of, you know, looked very interesting and also confusing for us, and we decided, okay, we would like to use our understanding of stochastic approximation theory to be able to explain what is happening over here, right? And the figure on the right explains what the greedy policy is at the end. So, in particular, this red curve corresponds to this trajectory, the green curve corresponds to this, and your blue curve corresponds to this trajectory. So what we are saying is that if you look at the greedy policy associated with phi theta n, where theta n is coming from this blue trajectory or is along the blue trajectory, and we look at the greedy policy.

So one can see that, you know, except for some initial few iterations where things jump around, pretty soon, the greedy policy is constant, and in the case of the blue trajectory, the greedy policy is actually pi star, which is the optimal policy, and this is what one would expect. On the other hand, for the green trajectory, the greedy policy is constant, but it is not pi star. So here, the greedy policy is very different. So what this means is that if you ran your algorithm,

And you know if in your run of your algorithm the behavior was according to this green trajectory then the optimal policy that you would end up figuring out based on the value of phi theta n could be very different from pi star itself. Now along the red trajectory we observed something very strange. So we observed that for along this trajectory the greedy policy that is associated with C theta n actually keeps jumping. In other words unlike this green and blue trajectory where the greedy policy is constant. In case of red trajectory, sometimes the greedy policy was some policy pi 0 and sometimes the greedy policy was pi 1 and it was never settling to either pi 0 or pi 1.

In other words, if you sort of take some theta over here, then phi theta's greedy policy will be pi 0 and if you allow the algorithm to run for few more iterations and look at the greedy policy associated with let us say some phi theta m plus m, that is the value of theta n plus m that is obtained after m iterations following theta n's estimate. So the greedy policy there could be different and we saw that this greedy policy kept jumping. In other words we observed some phenomena of policy oscillation. Now in this case you know it is not clear why it is oscillating between the two and what we also found surprising was that neither of the two policies that is neither pi 0 or pi 1 these are the two policies between which the greedy policies oscillating between neither of these policies is optimal. So if you run your algorithm that is this Q learning with linear function approximation and if it so happened that the trajectory turned out to be as you know predicted along this red trajectory then you know your greedy policy recall the greedy policies to be treated as an estimate of the optimal policy This estimate of the optimal policy will keep switching between pi 0 and pi 1 and surprisingly neither of pi 0 and pi 1 are you know the optimal policy right. So this was some you know toys experiment that we did but even in this toys experiment we saw some rich display of different asymptotic behaviors that is possible.

That is on different runs of the algorithm we saw different asymptotic behaviors and in certain cases we saw that the greedy policy associated with phi theta n was not stabilizing but it kept oscillating. And we wanted to understand can we give some explanation of why this is happening so that in case we understand why this is happening maybe we can provide a formal or principal fix to ensure that this does not happen. And I would like to emphasize that this oscillation and things like that have been repeatedly studied and it has also been studied that the behavior of Q learning in the function approximation context shows very diverse and strange behaviors. In particular, I encourage you to look at this paper titled Empirical Design in Reinforcement Learning. In particular, if you look at this figure over here, this corresponds to the



Empirical Design in Reinforcement Learning

Andrew Patterson ap3@ualberta.ca
Samuel Neumann sneuman@ualberta.ca
Martha White[1] whitem@ualberta.ca
Adam White[1] amw8@ualberta.ca
Department of Computing Science and Alberta Machine Intelligence Institute (Amii)
University of Alberta, Edmonton, Canada
[1]*Canada CIFAR AI Chair*

Editor: George Konidaris

Abstract

Empirical design in reinforcement learning is no small task. Running good experiments requires attention to detail and at times significant computational resources. While compute resources available per dollar have continued to grow rapidly, so have the scale of typical experiments in reinforcement learning. It is now common to benchmark agents with millions of parameters against dozens of tasks, each using the equivalent of 30 days of experience. The scale of these experiments often conflict with the need for statistical evidence, especially when comparing algorithms. Recent studies have highlighted how popular algorithms are sensitive to hyperparameter settings and implementation details, and that common empirical practice leads to weak statistical evidence (Machado et al., 2018; Henderson et al., 2018).

This manuscript represents both a call to action, and a comprehensive resource for how to do good experiments in reinforcement learning. In particular, we cover: the statistical assumptions underlying common performance measures, how to properly characterize performance variation and stability, hypothesis testing, special considerations for comparing multiple agents, baseline and illustrative example construction, and how to deal with hyperparameters and experimenter bias. Throughout we highlight common mistakes found in the literature and the statistical consequences of those in example experiments. The objective of this document is to provide answers on how we can use our unprecedente compute to do good science in reinforcement learning, as well as stay alert to pot pitfalls in our empirical design.

Keywords: Reinforcement Learning, Empirical Methodology

performance of variant of Q learning algorithm here in fact they work with DQN which is basically Q learning by using non-linear function approximation in particular they make use of neural network and hence the D over here stands for deep Q network and there is a mountain car there is an environment labeled mountain car right and there what they do is they run the Q learning algorithm for sufficiently large number of steps and right and at the end whatever policy that they obtain which is the policy that is greedy with respect to your Q value estimate that you obtain they sort of evaluate the value of that policy and on the X axis you have the range of the performance of that policy in some sense the average Q value estimate associated with the terminal policy that is there on the X axis and the Y axis somehow or in some sense shows the histogram or the density of how many times such policies were identified. So first of all you can see that there is a range of values from minus 500 to minus 300 right and one can see that you know if you run this algorithm it is very likely that your policy may end up with a value of minus 450 with very large probability. And, you know, in Q-learning, the hope is that we can find an estimate of the optimal policy.

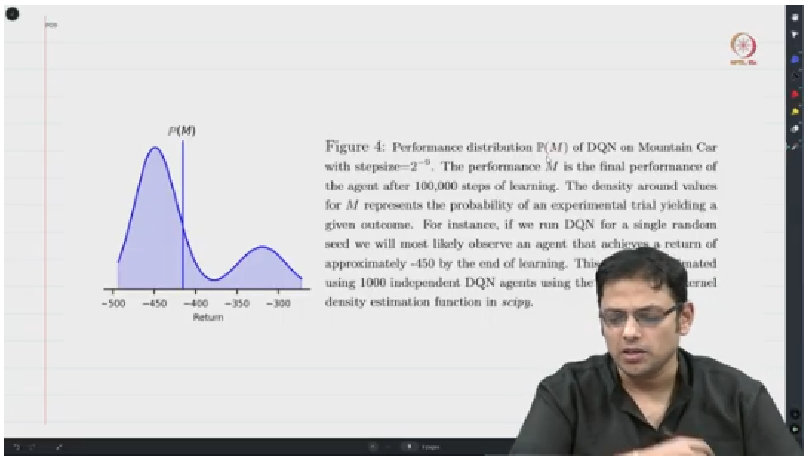


Figure 4: Performance distribution $\mathbb{P}(M)$ of DQN on Mountain Car with stepsize$=2^{-9}$. The performance M is the final performance of the agent after 100,000 steps of learning. The density around values for M represents the probability of an experimental trial yielding a given outcome. For instance, if we run DQN for a single random seed we will most likely observe an agent that achieves a return of approximately -450 by the end of learning. This [illegible]mated using 1000 independent DQN agents using the [illegible]ernel density estimation function in *scipy*.

But here it can be seen that on majority of the runs, your Q-learning algorithm will end up giving you a suboptimal policy, right? And only on few runs would you end up with a policy which has a very good performance, right? On the other hand, you can see there is a non-trivial probability that your Q-learning algorithm, right, will actually end up with a policy that is, performing quite worse right so this in some sense sets the stage for the next couple of lectures in which we will try to understand why this phenomena is occurring in particular we will also try to understand this phenomena of policy oscillation why it happens and also the fact that you know the Q learning algorithm somehow seems to be getting locked at different places you know in your search space which did not happen in the tabular Q learning case.

In the tabular Q-learning case, we were able to show that the Q-learning algorithm will almost always or almost surely converge to Q-star, which is the value function associated with the optimal policy. So, in the no-function-approximation case, the Q-learning algorithm actually does a very good job in finding the estimate of the optimal Q-value function. But surprisingly, when you move this algorithm to the function approximation setting, we end up with situations where, you know, there can be multiple limiting behaviors for the Q-value estimates—right? They can converge to different places, and on top of that, the greedy policy associated with these Q-value estimates could also keep oscillating. So, in the next couple of lectures, we will try to come up with an explanation for why this is happening, and, you know, hopefully, you can make use of that and try to build your new, improved Q-learning algorithm. So, until next lecture, thank you, goodbye, and Namaste.