The Effect of "Slowing Down" RL methods

Archit Gupta 150070001
Divya Raghunathan 150110086
Harshith Goka 150050069
Vishwajeet Singh 150050046

Abstract

We investigate the effect of reducing the frequency of decision making on the performance of batch reinforcement learning algorithms and policy gradient methods. We found that DI algorithms with a small value of d tends to outperform the vanilla algorithms

1 Introduction

Reducing the frequency of decision making in reinforcement learning algorithms can lead to less error propagation on account of fewer updates. However, there is a trade-off between decreasing the error and increasing the responsiveness of the agent. A "slowed down" algorithm would make the agent less responsive, which could lead to worse performance on examples in which the agent needs to adapt quickly. A very responsive agent would change its actions rapidly as it observes the environment, but the large number of updates it performs can lead to a larger error. The error in each update is significant when the MDP has noise or when function approximation is used. In problems with continuous state or action spaces, some form of function approximation is usually used. For such problems, reducing the frequency of decision making of the reinforcement learning algorithm could lead to better performance.
This project tries to establish experimentally that decision interval algorithms indeed perform better. The plots of the mean reward against the total episodes clearly supports the above arguments for the tested standard control tasks.

2 Related Work

The effect of the frequency of decision making on temporal difference learning was investigated in Siddharth [2017]. It was found that reducing the frequency of decision making leads to better performance of TD(λ) and Sarsa(λ) on problems with infinite state spaces, when generalisation over states is used. Previously, Durugkar et al. [2016]and Sharma et al. [2017] showed that repeating actions for more than one transition led to better performance of the DQN algorithm. In this project, we try to find out whether similar performance improvements can be obtained for batch reinforcment learning and policy gradient algorithms.

3 Decision Interval Algorithms

We define decision interval (DI) batch reinforcement learning algorithms and policy gradient algorithms analogously to decision interval Sarsa(λ) and decision interval TD(λ) defined in [1]. The decision interval d, $d >= 1$ is a measure of the frequency of decision making. Actions are selected at the start of each decision interval and the same action is used at subsequent states for the duration of the decision interval. Larger values of d indicate a lower frequency of decision making. When $d = 1$, the algorithm is the default algorithm. For $d > 1$, the agent is restricted to choose the same action for the duration of the decision interval.
A decision interval learning algorithm effectively tries to find the optimal policy of the induced MDP formed by skipping intermediate states within a decision interval. The transitions within a decision

interval are replaced by a new transition from the start state to the end state of the decision interval with a reward equal to the discounted sum of individual rewards received during the decision interval. The optimal policy of the induced MDP need not be optimal for the original MDP. However, decision interval algorithms may still perform better than the original algorithm on account of fewer erroneous updates.

3.1 DI Batch Reinforcement Learning Algorithms

The decision interval batch reinforcement learning algorithm (Decision interval Experience Replay), described in Algorithm 3, follows a policy derived from its current estimate of the action value function Q (ϵ greedy) for N episodes, storing a tuple (s, a, r_d, s') for each decision interval starting at state s and ending at state s', where $r_d = \sum_{t=t_1}^{t_1+d-1} r_t \gamma^{t-t_1}$. After N episodes, a batch update is performed by going through the set of stored tuples (the experience) M times, each time updating the current Q estimate.

Algorithm 1 Decision Interval Experience Replay

Input: decision interval d, number of episodes N, number of replays M, discount factor γ, learning rate α, exploration probability ϵ
e $\leftarrow 0$
experience $\leftarrow$ empty list
while e < total number of episodes **do**
  $t = 0$
  Initialize s
  Choose a uniformly at random
  $r_d \leftarrow 0$
  while episode not terminated **do**

    if $t \ \% \ d == 0$ **then**
      a $\leftarrow \epsilon$-greedy(s) start state $\leftarrow$ s, $r_d \leftarrow 0$
    end if
    Take action a, get reward r and next state s'
    $r_d \leftarrow r_d + r * \gamma^{t\%d}$
    if $t + 1 \ \% \ d == 0$ **then**
      end state $\leftarrow s$, Append (start state, a, r_d, end state) to experience, $r_d \leftarrow 0$
    end if
    $s \leftarrow s'$
    $t+ = 1$
  end while
  if e $\% \ N ==$ 0 **then**
    Update Q
  end if
  e += 1
end while

3.2 DI REINFORCE

Vanilla REINFORCE works by parameterizing the policy and doing a gradient ascent on the value function. Paszke et al. [2017] uses a neural network to parameterize the policy as a probability distributions over actions given a state. The exact formulation of slowing down with policy updates is depicted in Algorithm 2.

3.3 DI Q-Learning

Q-learning is an iterative algorithm which relies on a tabular representation of estimated Q values and updates it according to the reward seen and the $max_a Q(s', a)$ where s' refers to the next state. The algorithm has been described in Algorithm 3.

Algorithm 2 Decision Interval REINFORCE

// Assume θ as policy network parameters, global counter $i \leftarrow 0$
Input: decision interval d, number of episodes N, discount factor γ, learning rate α_{Adam}
Initialise policy network $\pi(s;\theta)$
Initialise network gradients $d\theta \leftarrow 0$
repeat
  $R_{DI} \leftarrow 0$
  states $\leftarrow$ list()
  rewards $\leftarrow$ list()
  Pick start state s_0 randomly
  $t \leftarrow 0$
  while s_t is not terminal **do**
    if $t\%d == 0$ **then**
      Choose a_t by sampling from the stochastic policy $\pi(s;\theta)$
      states $\leftarrow$ append(states, s_t)
    end if
    Take action a_t and observe reward r_t and next state s_{t+1}
    $R_{DI} \leftarrow R_{DI} + r_t(\gamma)^{t\%d}$
    if $(t+1)\%d == 0$ **or** s_{t+1} is terminal **then**
      rewards $\leftarrow$ append(rewards, R_{DI})
      $R_{DI} \leftarrow 0$
    end if
    $a_{t+1} \leftarrow a_t$
    $t \leftarrow t+1$
  end while
  $R \leftarrow 0$
  discountedRewards $\leftarrow$ list()
  for s, r in reverse(states), reverse(rewards) **do**
    $R \leftarrow r + \gamma R$
    discountedRewards $\leftarrow$ prepend(discountedRewards, R)
  end for
  $\overline{R} \leftarrow mean(discountedRewards)$
  $R_\sigma \leftarrow std(discountedRewards)$
  for s, r in reverse(states), reverse(discountedRewards) **do**
    Accumulate gradients wrt θ' :d$\theta \leftarrow d\theta + \Delta_{\theta'} \frac{r-\overline{R}}{R_\sigma}(log(\pi(s;\theta')))$
  end for
  Perform update of θ using $d\theta$
  $i \leftarrow i+1$
until $i > N$

4 Implementation Details

4.1 DI Batch Reinforcement Learning

We use nonlinear function approximation in the form of a neural network to represent Q, as the tasks we are testing on have continuous state spaces. A separate neural network to represent an older version of Q (Q_{old})is used, which is updated less frequently than the Q network. This makes the updates more stable. Both networks have one hidden layer with 64 nodes. The Q_{old} network is used to compute the target for the learning update, $r + max_a Q_{old}(s', a)$. The neural network is updated by training it on a sample of the experience, with the state action pair (s, a) as input and $r + max_a Q_{old}(s', a)$ as the target, where s is the start state of the decision interval, s' is the end state, a is the action taken and r is the net reward for the decision interval. For the CartPole task, the parameters N = 1000, M = 128, ϵ = 0.1 and α = 0.1 work well. All graphs plotted are averaged over 10 runs by randomizing the seed.

Algorithm 3 Decision Interval Q-Learning

$Q(s,a) \leftarrow 0 \forall s \in S, a \in A$
while $Q(s,a) \forall s \in S, a \in A$ has not converged **do**
  t $\leftarrow$ 0
  Pick start state s_0 randomly and choose start action ϵ-greedily from $Q(s_0, a)$
  while s_t is not terminal do **do**
    if t % d == 0 **then**
      Choose action a_t ϵ-greedily from $Q(s_t, a_t)$
      Take action a_t and observe reward r_t and next state s_{t+1}
      if $t > d$ **then**
        $Q(s_{t-d}, a_{t-d}) \leftarrow Q(s_{t-d}, a_{t-d}) + \alpha * (accum\_reward + \gamma^d * max_a Q(s', a)) - Q(s_{t-d}, a_{t-d}))$
      end if
      $accum\_reward \leftarrow r_t$
    else
      Take action $a_{t-t\%d}$ and observe reward r_t and next state s_{t+1}
      $accum\_reward \leftarrow accum\_reward + gamma^{t-t\%d} * r_t$
    end if
  end while
end while

4.2 DI REINFORCE

A neural network is used to parameterize the policy for the algorithm. There are 128 nodes in the single hidden layer of the network. For the entire episode, the neural network produces a probability distribution over the actions and an action is picked from it. The feature of automatic gradient differentiation in PyTorch is used to create a dynamic graph over the computations and backpropagate the loss function through the parameters. The learning rate of the algorithm is set using the Adam optimizer. For the CartPole task, the parameters N = 7000, γ = 0.99, ϵ = 0.1 and α_{Adam} = 0.01 work well. All graphs plotted are averaged over 10 runs by randomizing the seed.

4.3 DI Q-Learning

Q-Learning relies on a tabular representation of Q values. The state space for typical control tasks we have tested on have continuous state spaces and sometimes continuous action spaces too. We discretize the space using tile coding. For MountainCar task the state space is continuous but small so taking one tiling and 40 tiles for the entire state space gives decent results. This task defines three discrete actions viz. 0 (push left), 1(no push) and 2(push right), so these can be directly used as indices for the Q table stored using numpy arrays. Code and parameter values are taken from Alzantot [2017]. Instead of selecting the action ϵ-greedily we observed that taking actions with a softmax probability based on the Q values gives better results. The following parameter values are used : γ=1(undiscounted), ϵ=0.02, α=max(0.003,0.85^t). All graphs plotted are averaged over 10 runs by randomizing the seed. Training and testing of the agent has been done with the same value of DI.

5 Experimental Results

5.1 DI Experience Replay

The problem CartPole is moderately difficult compared to MountainCar. Here the state information consists of not just the position of the cart and the angle with vertical, but also the velocites of the cart and pole (at tip). DI experience replay with DI 2 and 3 perform better than regular experience replay as can be seen in Figure 1. Very large values of DI hardly improved with time. DI = 2 performed the best, followed by DI = 3. This seems to indicate that small values of the decision interval work better than the vanilla algorithm with DI=1. Putting a large d=9,11,13 hampers the learning of the agent and results in decreased responsiveness of the agent to actual scenario.

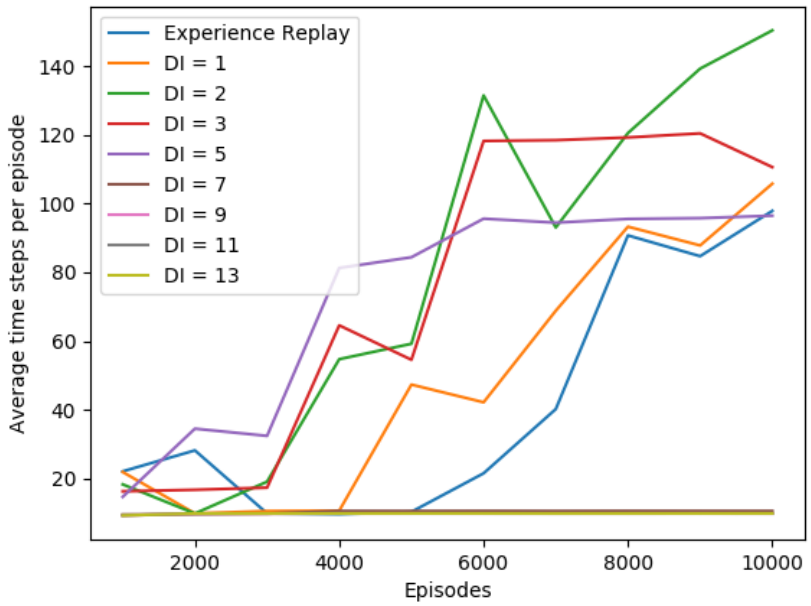

Figure 1: DI Experience Replay on CartPole

5.2 DI REINFORCE

Refer to Figure 2 for the plot

5.3 DI Q-Learning

MountainCar which is a relatively simple task compared to CartPole shows interesting trends as can be seen in Figure 3. The vanilla Q-learning tops out with a mean reward of around -130 after 10000 iterations. Large DI like d=7,10 are slightly better than this, but with small d=2,3,5 we able to achieve -118 as the mean reward. A surprising result is obtained with d=30 when the agent learns to get mean reward of -105. The hyposthesis that larger DI will always give better mean is refuted by d=55 where the agent is not able to learn much.
A possible explanation for the observations is that with smaller values of d the agent benefits because the updates are made faster and more accurately for every dth state. In this task the state space is actually quite simple. A simple human strategy would be to accelerate to right till a certain point, then accelerate to the left till a large height, and then go right using the force of gravity to your advantage. Thus there is no requirement for the policy to be different with respect to neighbouring states.
Following the same action for d steps thus benefits the agent. A steeply large reward for d=30 could be because the number of states till an alternate action is to be chosen is a multiple of the value of d chosen, or that the actual distance moved between states is not very large. In general we want to choose a suitable decision interval which is not too large otherwise we risk the agent not learning anything at all

6 Conclusion and Future Work

In this project we establish empirically that in general slowing down of batch reinforcement learning algorithms, Q-learning and policy gradient algorithms seems to improve the performance of the agent. This idea thrives on the underlying fact that in real MDPs we do not need to responsive with respect to each transition, and also propagating less noise in successie updates.
Additionally, they take lesser time to execute especially if the update step is computationally expensive, which happens with a frequency inversely proportional to the decision interval.

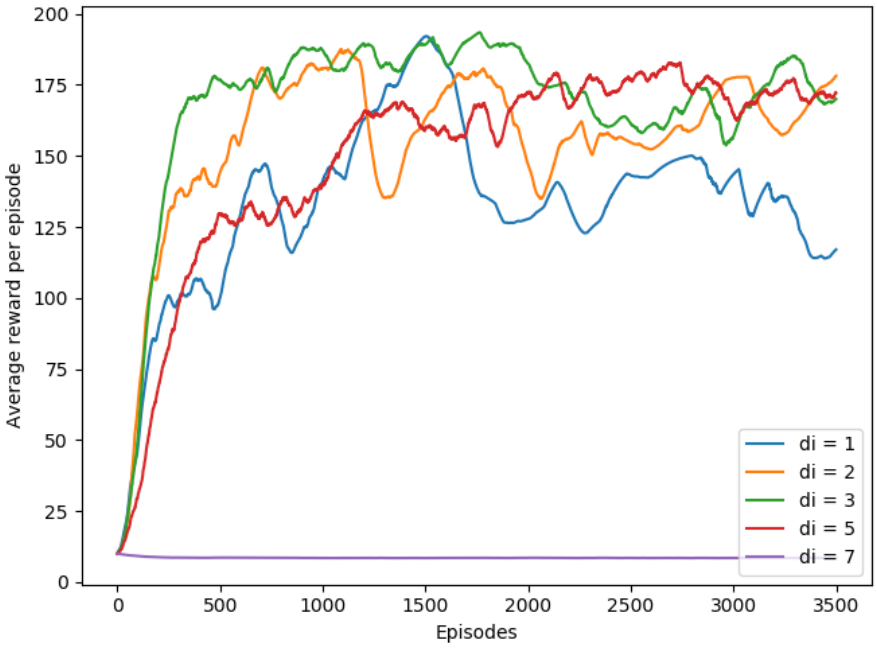

Figure 2: DI Reinforce on CartPole

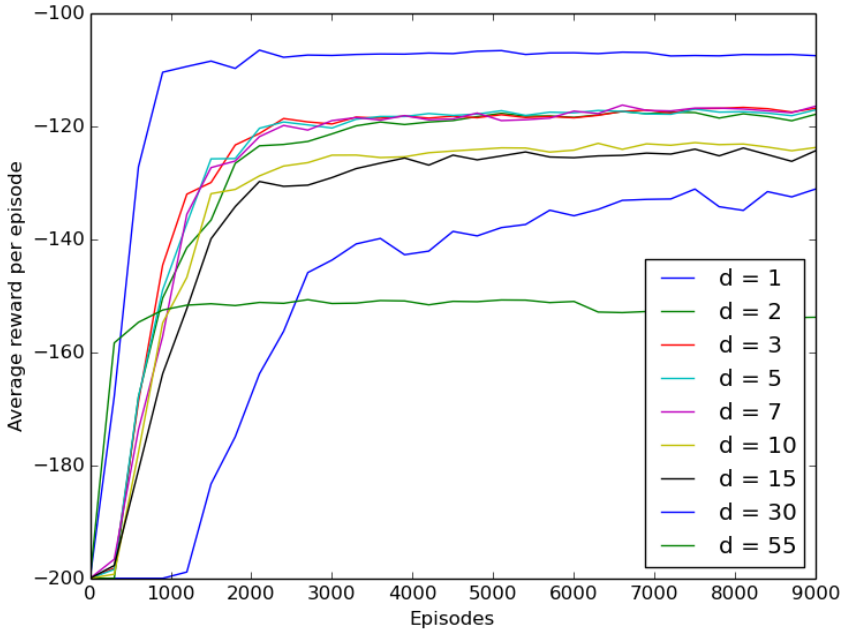

Figure 3: DI Q-Learning on MountainCar

7 Future Work

We experimentally showed the superior performance of slowed-down RL metohds, but do not give any theoretical backing for it. That could be a good path for further research in this topic. We have only looked at agents which keep a fixed value for DI throughout the course of learning. It could be possible that changing it during the run, somewhere on the lines of decaying learning rate and other parameters could further improve performance.

References

A. Siddharth. On the effect of the frequency of decision making in temporal difference learning. Master's thesis, Indian Institute of Technology Bombay, 2017.

Ishan P Durugkar, Clemens Rosenbaum, Stefan Dernbach, and Sridhar Mahadevan. Deep reinforcement learning with macro-actions. *arXiv preprint arXiv:1606.04615*, 2016.

Sahil Sharma, Aravind S Lakshminarayanan, and Balaraman Ravindran. Learning to repeat: Fine grained action repetition for deep reinforcement learning. *arXiv preprint arXiv:1702.06054*, 2017.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

Moustafa Alzantot. Solution of mountaincar openai gym problem using q-learning, 2017. Avaialble at `https://gist.github.com/malzantot/9d1d3fa4fdc4a101bc48a135d8f9a289#file-mountaincar_qlearning-py`.