Overcoming forgetting in Batch-RL

Shriram S B
150050099

Abhijeet Awasthi
174050004

Divyansh Pareek
150050029

Abhinav Goyal
150050108

Abstract

We consider the task of estimating Q-function using Neural Nets as tools of function approximation, by experience replay in Batch Reinforcement learning setting. Based on lecture discussions, we believe that learning the network in this setting could be forgetful if the data generated by the given policy is replayed a large number of times. The larger number of replays might drift the learned parameters of the Q-network towards the most recently seen set of experiences, and this set of converged parameters might not represent the correct Q-values for the other sets of observed experience. We propose a way to allow learning in a Batch-RL setup to be less forgetful, which involves associating an importance value with each parameter of the network and imposing a penalty on the shift of highly important parameters.

1 Introduction

We consider learning the Q-function using neural networks as a tool for function approximation. In the Batch-RL setting during experience replay, data presented to the Q-network will typically contain sets of samples from different state-action pairs from the underlying environment. Q-estimates obtained using this data from a batch of episodes might differ from the Q-estimates obtained using data from another batch of episodes. If these two episodes contain experiences from different state localities of the environment, then they potentially drive the parameters of the Q-network towards two different sets of optimal parameter values, neither of which is likely to be the optimal parameter value set which will best represent the Q-values based on the combined experiences. This is likely to cause the parameters learned from the data of one batch of episodes to change drastically when updated to learn from data of another batch of episodes.

To alleviate the above problem, we consider recent advances in methods addressing catastrophic forgetting in neural nets. In particular, we apply an importance weighted parameter regularization based approach proposed by [1] to avoid forgetting across experience replays in Batch-RL setting. We observe that our method converge to decent estimate of Q function more quickly as compared to methods which do not have a provision to avoid forgetting.

2 Previous Work

When neural networks are trained sequentially on different tasks, the parameters learned by back-propagating errors adapt to optimize for the most recent task. If this parameter learning is unconstrained, this tends to renders poor performance on the previously learned tasks. This problem is widely referred to as the problem of Catastrophic Forgetting in neural networks [2]. [2] [1] [3] are a few recent works which introduce ways to constrain the weights of a neural network so that sequential task learning is less forgetful. All the three works share a common idea of calculating

importance of each parameter w.r.t. a given task T_t. When learning a new task T_{t+1}, the parameters that are previously seen as important, are shielded from a significant change. The so-far unimportant parameters are utilized to learn new tasks. This helps in preserving performance of the network on previous tasks while allowing them to learn a new task. [2] computes parameter importance by using fisher information, [1] proposes an interesting approach of computing line integrals over a given learning trajectory. These line integrals try to consider contribution of a given parameter in minimizing the loss function. Similar to [2], [3] proposed to compute parameter importance by computing partial derivative of l_2 norm of the final layer representation learned by a network, w.r.t. its parameters. All the three methods estimate importance (Ω_k) of a parameter θ_k, by using sensitivity of either the loss function or the learned representations w.r.t. the network's parameters.

In general, while learning the task T_{t+1} with objective $\mathcal{L}_{t+1}$ we modify the learning objective according to Equation 1.

$$\tilde{\mathcal{L}}_{t+1} = \mathcal{L}_{t+1} + \lambda \sum_k \Omega_k^t \left(\theta_k - \theta_k^t\right)^2 \tag{1}$$

$$\theta_k^{t+1} = \underset{\theta_k}{\operatorname{argmin}}\ \tilde{\mathcal{L}}_{t+1} \tag{2}$$

The term in RHS of Equation 1 penalizes parameters (θ_k) to change from their values obtained after learning previous tasks. This is similar to l_2-regularization, except that penalties are weighted by importance of parameters (Ω_k^t) observed in previous learning experiences. Ω_k^t represents importance of parameter θ_k in learning upto task T_t. λ is a hyperparameter similar to weight decay parameter in l_2-regularization. Updated parameters after learning task T_{t+1} are given according to Equation-2.

We briefly review the method proposed by [1] as an example to compute Ω_k^t. Using first order optimization they show that while learning a task T_t, contribution of a parameter θ_k in decreasing the loss $\mathcal{L}_t$ is proportional to negative of product of parameter update ($\theta_k^{'}$) and gradient (g_k) of the loss $\mathcal{L}_t$ w.r.t to the parameter θ_k. The line integral of this quantity over the learning trajectory gives the overall contribution of the parameter in decreasing the loss (over training data) during the learning process. Further, if two parameters contribute the same amount to the decrease of loss, the parameter that changed less during the course of learning trajectory is considered to be more important for the task being learned (due to high sensitivity w.r.t. loss function). Ω_k^t is thus obtained according to Equation 3 and Equation 4.

$$\omega_k^t = -\int_{\tau=t-1}^{\tau=t} g_k^t(\theta(\tau))\theta_k^{'}(\tau)d\tau \tag{3}$$

$$\Omega_k^t = \sum_{s \leq t} \frac{\omega_k^s}{(\Delta_k^s)^2 + \xi} \tag{4}$$

Here τ refers to a parameter update step in learning trajectory of task T_t. Δ_k^s represents change in model parameter θ_k due to learning of task T_s. ξ is a small constant to avoid division with zero.

3 Avoiding forgetting in Batch-RL

We use Batch Reinforcement learning to learn Q values from experience generated by interacting with an MDP. Batch Reinforcement Learning Algorithm is outlined in Algorithm 1. We aim to alleviate the problem of forgetfulness in a Batch-RL setting by regularizing the parameters of our Q-network using the method proposed by [1]. We treat each iteration of experience replay routine as a separate task.

Consider any iteration (say iteration $t+1$) of $trainBatchExperienceReplay$ in step 19 of Algorithm 1. This iteration is treated similar to task T_{t+1} in previous section. $\mathcal{L}_{t+1}$ is given according to Equation 5. We try to minimize the mean square error between Q values represented by the network and the targets formed by 1-step bootstrapping, over all state action pairs. Using Equation 3 and Equation 4 we first identify the Q-network parameters which play an "important" role in learning from the dataset generated during iteration t. Then, we constrain these "important" parameters from any significant changes while learning from a dataset generated during the next iteration $t+1$, according

to Equation 1.

$$\mathcal{L}_{t+1} = \Sigma_{di \in D} \Sigma_{a_i \in A} (Q(s_i, a_i) - (r_i + \gamma \max_a Q(s_{i+1}, a))^2) \tag{5}$$

This is actually not the accurate loss function as in each iteration, we evaluate the network to get the target $max_a Q(s_{i+1}, a)$. Still, just the knowledge of the MSE loss in the current batch in line 4 of algorithm 2 is enough to update the importances of the parameters. We expect this way of constraining parameters to help us prevent the drift of model parameters towards the most recently played data. [2] refer to this way of regularising parameters as Elastic Weight Consolidation (EWC).

Algorithm 1 Batch Reinforcement Learning

1: $Q \leftarrow Q_0$ ▷ Initialize action value function
▷ Generate batches of experiences, and update policy after each generation
2: **repeat**
3: $D \leftarrow \phi$ ▷ Initialize sequence of experiences D
4: $i \leftarrow 0$ ▷ Initialize size of D
▷ Collect experiences for m episodes
5: $episodes \leftarrow 0$
6: **repeat**
7: $i \leftarrow i + 1$
8: **if** new episode **then**
9: $s_i \leftarrow getStateFromEnvironment()$
10: $episodes \leftarrow episodes + 1$
11: **end if**
12: $a_i \leftarrow selectAction(Q, s_i)$
13: $Execute\ a_i$
14: $r_i \leftarrow getRewardFromEnvironment()$
15: $s_{i+1} \leftarrow getStateFromEnvironment()$
16: $d_i \leftarrow (s_i, a_i, r_i, s_{i+1})$
17: $D.append(d_i)$
18: **until** episodes == m
19: $Q \leftarrow trainBatchExperienceReplay(D)$ ▷ Update Policy
20: **until** Q has converged

Algorithm 2 trainBatchExperienceReplay(D)

1: $Q \leftarrow Q_0$ ▷ Initialize action value function
2: **for** $iteration = 1\ to\ k$ **do** ▷ Train for k iterations
3: **for all** $i \in [1..[D]]$ **do** ▷ Replay each experience. $d_i = (s_i, a_i, r_i, s_{i+1})$
4: $\mathbf{w} \leftarrow \mathbf{w} + \alpha \nabla L_i(\mathbf{w})$ ▷ L_i is the regularized loss for i^{th} example
5: **end for**
6: **end for**

We propose that this idea of Elastic weight consolidation (EWC) should be helpful when the Q-values are parameterized and learnt using batch RL. Following can be considered as main contributions of our method.

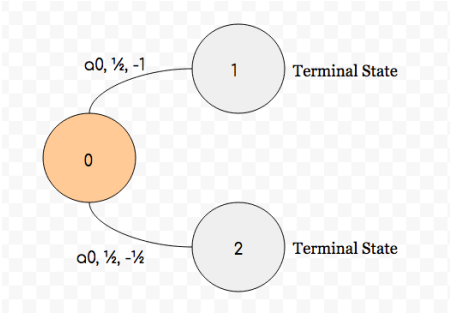
1. A naive way to avoid forgetting is to replay the union set of all the data generated in past during experience replay. But, considering MDPs with large state, action pairs and having longer trajectories, it is not feasible to store all the data generated in past. Hence we believe that out method is memory efficient as compared to the naive method.
2. We expect our method to converge faster to a decent approximation of Q function, as learning is less forgetful which helps in retaining approximations learned during previous replays.
3. Fewer samples are required in each iteration of experience replay. Suppose, samples in t^{th} and $(t+1)^{th}$ iteration are such that for any two samples (s_i, a_i, r_i, s_{i+1}), one from t^{th} iteration and another from $(t+1)^{th}$ iteration, (s_i, a_i) do not match. Without EWC, it is not

possible to learn Q without forgetting values of Q for (s_i, a_i) in t^{th} iteration, as gradient descent will optimize for the new dataset, completely moving away from the optimal value of the old dataset. It would require lots of samples which cover all parts of the state space to avoid forgetting. However, with EWC, during iteration $(t+1)$, it most likely modifies only the less important parameters for iteration t, thus learning Q-values for another part of state space along with retaining the Q-values for part of state space corresponding to iteration t(which it has previously seen).

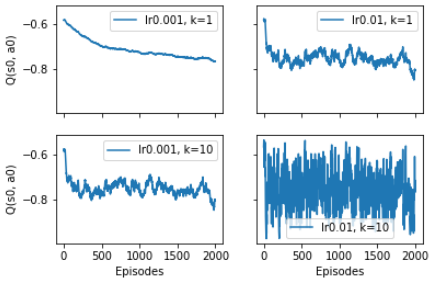
4. Suppose, some samples, (s_i, a_i, r_i, s_{i+1}), from iteration t and $(t+1)$ match in (s_i, a_i), but have different (r_i, s_{i+1}) due to stochasticity of the environment. It is most likely that both iterations, t and $(t+1)$ do not have enough samples to model the correct distribution of the next state and reward from (s_i, a_i). Without EWC, Q values in iteration $(t+1)$ will move towards optimality of current dataset and completely forget old values of Q, thus optimizing incomplete distribution. EWC enforces a balance between target values of the input (s_i, a_i) calculated from datasets from iteration t and $(t+1)$. This helps to reduce oscillation of Q values and intuitively leads the network parameters to optimize for target values of (s_i, a_i) across multiple experience sets (multiple tasks in the catastrophic forgetting setting).

4 Experiments

Ideally, in algorithm 1, we would like to have large m and in algorithm 2, we would like to have large iterations k. Large m helps to get a good model of the environment. Large k makes the best use of the available data. Large k helps to get to optimal Q-value without lot of iterations of collecting episodes. But since m is finite, having k to be very large overfits to the dataset which is not representative of the environment. This causes oscillations in the Q-values predicted by the network. Figure 1



(a) MDP with 3 states, 1 action. State 0 has equal probability of moving to state 1 or state 2 on action $a0$. Rewards are -1 and -1/2 respectively. Q($s0$, $a0$) = -0.75.



(b) Q-values while training on this mdp.

Figure 1: Stochastic MDP demonstrating Forgetting using Batch-RL

To better demonstrate the effect of limited data due to small m, we use the MDP in Figure 1 with 1 episode per batch. So, each batch will have only one transition. We try to learn only Q($s0$, $a0$), ie only for non-terminal states. Graph in Figure 1 shows graphs of varying α (lr) and k in algorithm 2. Small

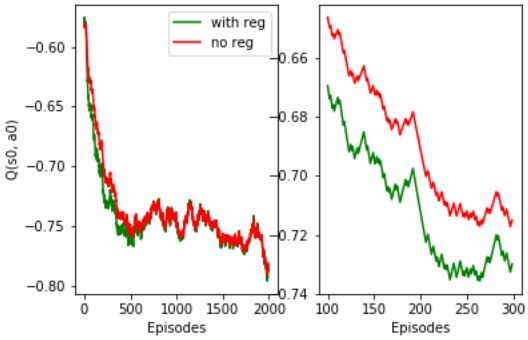
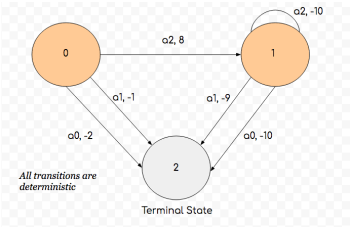


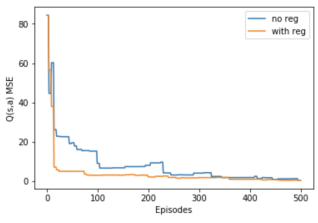
Figure 2: Comparing learning curve while training on Figure 1 with and without weight regularization. Right - zoomed in version of Left

α and k causes less oscillations but takes a lot of episodes to get to the true Q value. On the other hand, increasing α or k causes the algorithm to start oscillating. This is because it is equally likely to get a target of -1 or -0.5 in every batch of training. Choosing large k will make the parameters change such that the output for Q($s0$, $a0$) is the recent target (-1 or -0.5, whichever it got in the most recent batch). Figure 2 shows the learning curve with weight regularization. We tuned α and k in case without regularization to so that its amount of oscillations are similar to that of the case with regularization. We consider reaching in the range [-0.8, -0.7] as the criteria for converging. It is clear that our method of weight regularization converges faster (160 episodes) than batch-RL without weight regularization (206 episodes). The neural network had only one parameter (b) to learn, which approximated the true Q($s0$, $a0$). Without weight regularization, the parameter moves to the value of the reward recently obtained (either -1 or -0.5 without any constraints). With regularization, the learning of the parameter is also constrained from previous values of the parameter. So, suppose the previous value of the parameter was around -0.73 and a reward of -1 is obtained in the current episode, while learning in the current batch, the loss function will be $(-1-b)^2 + \lambda * i_b * (-0.73-b)^2$. This causes the parameter b to not move much away from -0.73, hence helping to converge better. Here, $*i_b*$ is a parameter that decays the importance values from previous tasks. We also decayed importance of paramters calculated from old tasks in Eqation (4) by a hyperparamter as newer tasks arrive. This helps to keep ω_k^t from shooting up to ∞ as t increases and also helps to learn in cases where transition probabilities are varying.

Having described in detail the effect of regularization on smaller MDP, we show the results on large MDP. Figure 3 shows oscillations occuring for different k and α in algorithm 2. Figure 4 compares the learning curves for the training with regularization against without it. 2-layer neural network with 10 hidden neurons has been used. Here as well, it can be observed that method with regularization converges faster.



(a) Deterministic MDP with 3 states and 3 actions



(b) Learning curve while training

Figure 5: A 3 state deterministic MDP and corresponding learning curves when we train without any weight regularization and with weight regularization

Further to compare both the methods, with and without weight regularizations, we train our agent on the MDP shown in Figure 5(a). The corresponding learning curves in Figure 5(b) show that the agent

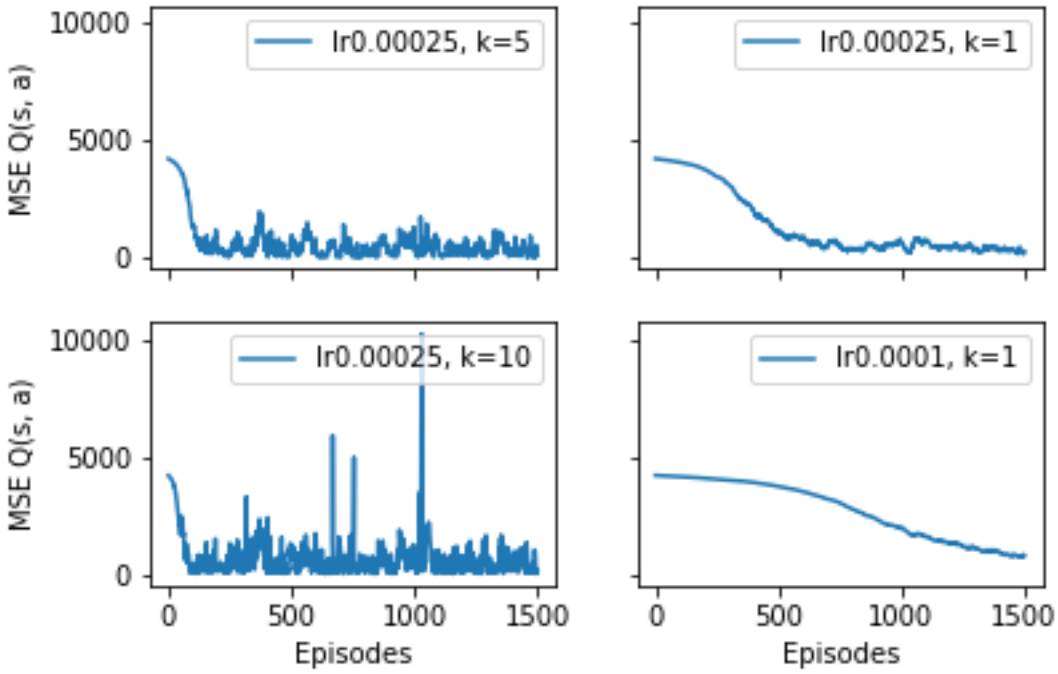


Figure 3: Training curve with no regularization with varying parameters on MDP with 10 states, 4 actions with random transition probabilities. Reward are random in the range [-20, 0]

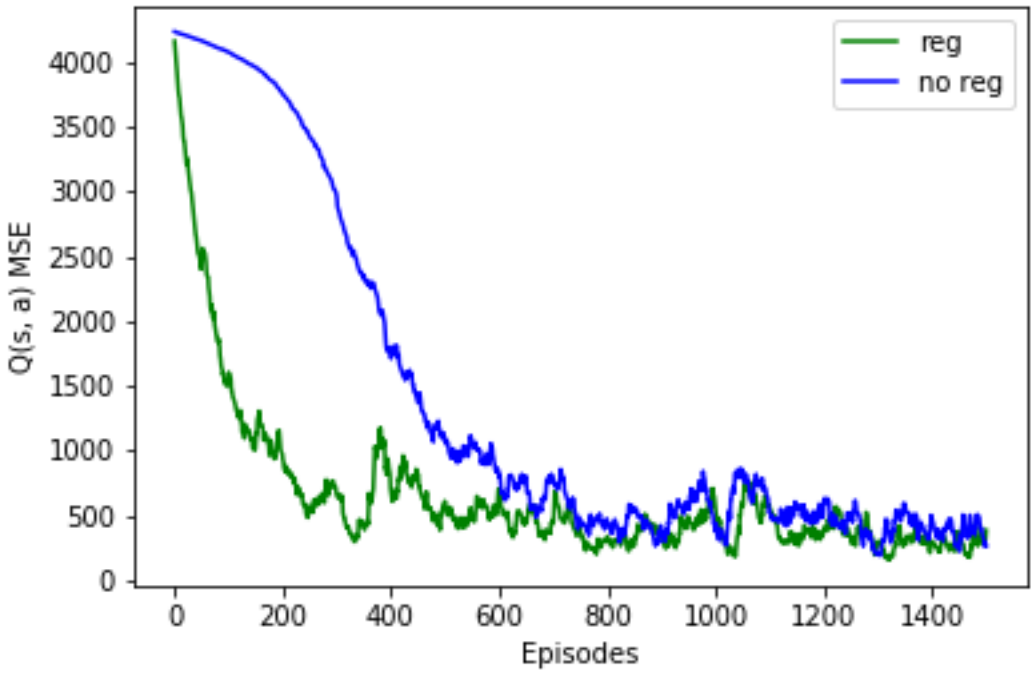


Figure 4: Comparing with and without regularization on same MDP as in Figure 3. Graph compare MSE with the true Q-value for all non-terminal states.

trained with weight regularizations converges faster whereas the one without trained without any weight regularizations takes more than around 220 episodes to get similar Q-value estimates. This again demonstrates that adding weight regularizations help converge faster in Batch-RL setting as it adds stability to the Q-estimates the Q-network predicts.

5 Conclusions & Future Work

Through our experiments, we have demonstrated that approximating Q functions using neural networks is indeed forgetful in the Batch-RL setting. We have shown improvements by using the EWC method to retain the Q values learned in preceding iterations of experience-replay. Through these experiments, it is clear that there is considerable scope for the application of such methods in this setting.
As future work, it would be interesting to see the gains of applying the EWC method on more complex environments than the ones presented. Settings that have higher state and action spaces are natural to follow, examples being Keepaway Soccer from [4]. Another nuance that we observed during our experiments is that the gains that EWC offers are sensitive to the parameter λ that weigh the penalty fo the change of parameters to the original loss for learning. It would be interesting to ponder about methods that could potentially make the method more robust.

References

[1] Friedemann Zenke, Ben Poole, and Surya Ganguli. Continual learning through synaptic intelligence. In *International Conference on Machine Learning*, pages 3987–3995, 2017.

[2] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.

[3] Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory aware synapses: Learning what (not) to forget. *CoRR*, abs/1711.09601, 2017.

[4] Shivaram Kalyanakrishnan and Peter Stone. Batch reinforcement learning in a complex domain. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, page 94. ACM, 2007.