# 048716 - Advanced Subjects in RL - Final Project
# Shallow Updates for Deep Reinforcement Learning

Tal Daniel

February 2019

## 1 Paper Summary

In this work, we present a short summary of [10], and our own personal addition to this paper. The paper begins with a brief overview of the Reinforcement Learning field and defining two types of *function approximators* in order to tackle the *curse of dimensionality* problem: using deep neural networks (Deep RL or DRL) and using linear architectures, e.g. linear regression (Shallow RL or SRL). The paper makes use of two popular methods that are considered SRL methods, where both are *batch* algorithms based on a least squares (LS) approach: Least Squares Temporal Difference (LSTD) and Fitted-Q Iteration (FQI). Both methods are known to be stable and data efficient, but their performance is highly dependent on the quality of the feature representation, and in practice, finding such representation is proven to be a difficult task. Thus, using deep architectures such as deep neural networks can help with learning powerful representations which can be used as features for the SRL algorithms.

The paper suggests a new *hybrid* approach the combines batch SRL algorithms with online DRL, mainly tailoring LSTD and FQI (SRL) on the last hidden layer of a Deep-Q Learning Network, or DQN (DRL), by retraining the last hidden layer's weights. This hybrid algorithm is called Least Squares DQN (LS-DQN).

### 1.1 SRL algorithms

#### 1.1.1 Least Squares Temporal Difference Q-Learning (LSTD-Q)

LSTD and LSTD-Q are batch off-policy SRL algorithms. LSTD-Q learns a control policy $\pi$ from a batch of samples by estimating a *linear approximation* $\hat{Q}^{\pi} = \Phi w^{\pi}$ of the action value function. We define $w$ as the a set of weights and $\Phi$ as a set of features. The weights are learned by enforcing $Q$ to satisfy a fixed-point equation w.r.t. the projected Bellman operator. Given $N_{SRL}$ samples, we can approximate the system of linear equations as follows:

$$\tilde{A} = \frac{1}{N_{SRL}} \sum_{i=1}^{N_{SRL}} \left[ \phi(s_i, a_i)^T \big( \phi(s_i, a_i) - \gamma \phi(s_{i+1}, \phi(s_{i+1})) \big) \right]$$

$$\tilde{b} = \frac{1}{N_{SRL}} \sum_{i=1}^{N_{SRL}} \left[ \phi(s_i, a_i)^T r_i \right]$$

The weights can be calculated by a least squares minimization: $\tilde{w}^{\pi} = argmin_w ||\tilde{A}w - \tilde{b}||_2^2$ or by computing the pseudo-inverse: $\tilde{w}^{\pi} = \tilde{A}^{-1}\tilde{b}$.

### 1.1.2 Fitted Q Iteration (FQI)

The FQI algorithm is also a batch SRL algorithm that computes iterative approximations of the Q-function using regression. In the regression task, the supervised target in FQI is defined as: $y_i = r_i + \gamma max_{a'} Q^{N-1}(s_{i+1}, a')$. Then, by minimizing the MSE loss:

$$Q^N = argmin_Q \sum_{i=1}^{N_{SRL}} \big(Q(s_i, a_i) - y_i\big)^2$$

In the linear approximation form, we get the following:

$$\tilde{A} = \frac{1}{N_{SRL}} \sum_{i=1}^{N_{SRL}} \big[\phi(s_i, a_i)^T \phi(s_i, a_i)\big]$$

$$\tilde{b} = \frac{1}{N_{SRL}} \sum_{i=1}^{N_{SRL}} \big[\phi(s_i, a_i)^T y_i\big]$$

The DQN algorithm can be viewed as the online form FQI.

## 1.2 DRL algorithms

### 1.2.1 Deep Q-Network (DQN)

The DQN is an off-policy algorithm that learns the Q function by minimizing the mean squared error of the Bellman equation. The DQN consists of two separated neural networks, namely the current network with weights $\theta$ and the target with weights $\theta_{target}$. By fixing the target network, the algorithm is similar to FQI. The DQN network, as part of being off-policy, is optimized using an Experience Replay (ER) Buffer, which holds the agent's experience (tuples of states, actions, rewards and next states). Using ER provides improved performance and stability.

### 1.2.2 Double DQN (DDQN)

DDQN is a modification to the original DQN, aiming to address the overly optimistic estimates of the value function (by calculating the TD target, how are we sure that the best action for the next state is the action with the highest Q value?) as follows: the target update $y_t = r_t$ if $s_{t+1}$ is terminal (as the original), and $y_t = r_t + \gamma Q_{\theta_{target}}\big(s_{t+1}, argmax_a Q_\theta(s_{t+1}, a)\big)$ where the action selection is performed with the current network the evaluation is done with the target network.

## 1.3 The Novelty - The LS-DQN Algorithm

The state-of-the-art of this paper is the Least Squares DQN algorithm which implements the hybrid approach between SRL and DRL by periodically re-training the last hidden layer with an SRL method. We define the following:

- $N_{NDRL}$ - number of DRL training steps (before an SRL step
- $N_{SRL}$ - number of samples (batch size) to be used in the SRL step
- $w_k^{last}$ - the weights of the last hidden layer
- $\phi(s)$ - the features extracted from the last hidden layer by performing a forward pass up to that layer
- $\phi(s, a)$ - the *augmented* features

As the features extracted from the last hidden layer are only state-dependent, and the SRL algorithms use features which are functions of state and actions, we define the augmentation $AUG(\phi(s)) = \phi(s,a)$ as the zero-padding to a one-hot state-actions feature vector. That is, $\phi(s,a)$ is $\phi(s)$ on a subset of indices the belong to action $a$ and zero otherwise.

**Algorithm 1** LS-DQN Algorithm

**Require:** $w_0$

**for** $k = 1 \cdots SRL_{iters}$ **do**
  $w_k \leftarrow$ trainDRLNetwork$(w_{k-1})$ ▷ Train the DRL network for $N_{DRL}$ steps
  $w_k^{last} \leftarrow$ LS-UPDATE$(w_k)$ ▷ Update the last layer weights with the SRL solution
**end for**

**function** LS-UPDATE$(w)$
  $D \leftarrow$ gatherData$(w)$
  $\Phi(s,a) \leftarrow$ generateFeatures$(D, w)$
  $w^{last} \leftarrow$ SRL-Algorithm$(D, \Phi(s,a))$
  **return** $w^{last}$
**end function**

## 1.4 The Novelty - Regularization

It was found that using off-the-shelf SRL algorithms (FQI, LSTD-Q) resulted in instability and degradation of the DRL performance. The cause is presumably the 'slow' SGD computation in DRL that retains information from older training epochs, while vanilla SRL methods ignores data from previous batches. Thus, a novel regularization is suggested to improve the performance. We now introduce a Bayesian regularization method for LSTD-Q and FQI the uses the last hidden layer's weights as a prior for the SRL algorithm. The final equation for the FQI algorithm:

$$w^{last} = (\tilde{A} + \lambda I)^{-1}(\tilde{b} + \lambda w_k^{last})$$

## 1.5 Experiments and Analysis

We present 3 types of experiments done in this paper.

### 1.5.1 SRL Algorithms with High Dimensional Observations

This type of experiment is meant to asses the performance of SRL algorithms in domains with high dimensional observations. The following procedure is performed:

1. Train DQN agents on two Atari games (Breakout and Qbert) using the vanilla DQN. Periodically save the current network's weights and the current Experience Replay.
2. Use an SRL algorithm to re-learn the weights of the last layer.
3. Evaluate the resulting DQN network by temporarily replacing its weights with the SRL solution weights. After the evaluation the original weights are replaced back, The evaluation process entails 20 roll-outs with an $\epsilon$-greedy policy.

Results and Insights:

- **Regularization** - without it, results were poor. It is assumed to stem from the sparseness of the features due to the ReLU activation of the hidden layer which causes the approximated matrices to be ill-conditioned. Two regularizers were used: $l_2$ and BP where $\lambda \in [0, 100]$. BP performed better across domains and were less sensitive to the scale of $\lambda$.It is recommended to use $\lambda \in [0.1, 10]$
- **Data Gathering** - two approaches were considered: generating new data from the current policy and using the ER. Poor performance was observed when using new generated data in comparison to using the ER. It is believed that the reason is that the ER contains data sampled from multiple policies and thus exhibits more exploration of the state space.

- **Policy Improvement** - performing multiple iterations on the same dataset did not improve the results. Possible explanation is that by improving the policy, it reaches new areas in the state space that are not represented well in the current ER, and therefore are not approximated well by the SRL solution and the current DRL network.

#### 1.5.2 Atari Experiments

The next set of trials is testing the performance on five Atari games: Asterix, Space Invaders, Breakout, Breakout, Q-Bert and Bowling. Both SRL algorithms tested with both DQN and DDQN. The Wilcoxon signed-rank test was also done to measure how related samples differ in their means. Results and Insights:

- The LS-DQN attained better performance compared to the vanilla DQN agents.
- For the game Asterix, the DQN's score crashes to zero at some point, while the FQI version of LS-DQN solved this issue.

#### 1.5.3 Ablative Analysis

This set of experiments tries to give explanation to the improved performance of the LS-DQN algorithm, focusing on the FQI variation. At each LS-UPDATE step, two types of optimizations are made:

1. FQI with a Bayesian prior and a LS solution.
2. ADAM optimizer with and without Bayesian regularization term in the loss function.

Different mini-batch (MB) sizes of 32, 512 and 4096 are tested, while $\lambda = 1$. It is worth noting that ADAM has more hyper-parameters to tune than FQI (as it is using the whole dataset). Results and Insights:

- Larger mini-batches resulted in improved performance.
- The LS solution outperformed the ADAM solutions for the most part.
- Solutions with prior perform better than solutions without.
- Not only large batch methods perform better, but are also easier to tune (less hyper-parameters).

In addition to the above, LS solution weights are close to the baseline DQN and the distance between the weights is inversely proportional to the performance of the solution. That is, best performance using FQI is achieved when the solution was the closest to the DQN solution (distance is measured in the $l_2$ norm).

### 1.6 Related Papers

This paper is the first one to successfully combine DRL with SRL, although using the last hidden layer of a DNN as a feature extractor has been considered in the context of transfer learning [2]. As for applying regularization in RL, it was recently introduced in problems that combine value-based RL with other learning objectives, like supervised learning from expert demonstration [3]. SRL algorithms perform well with regularization [4]. Recent works that cited this paper include uncertainty output of the last layer in a Bayesian deep Q network [1].

### 1.7 Conclusion

This paper presents a hybrid approach of DRL with SRL that is proven to be superior to the vanilla DQN in the Atari domain. In the next section, we will how this approach can be extended with another SRL algorithm.

# 2 Boosted LS-DQN, Boosted LS-Dueling-DQN and LS-DDPG

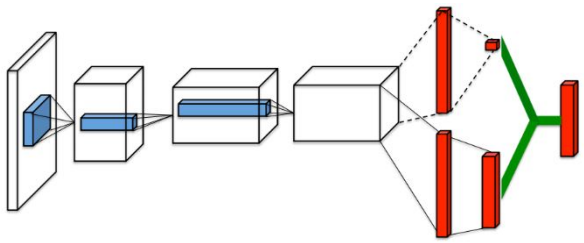
In this section, we present our extension to the paper [10] in the form of introducing a modification to the FQI algorithm called Boosted-FQI with the vanilla DQN, and also combining it with Dueling DQN architecture which has been proven to perform better than the vanilla version [9].

Our contribution in this project is as follows:

- Modern implementation of LS-DQN in PyTorch (the original one is in LUA).
- Testing the algorithm on new Atari environments: Pong and Boxing.
- Integration of the algorithm with Dueling DQN and Double DQN [9, 8].
- Introducing new SRL algorithm for the LS-DQN: Boosted FQI - Boosted LS-DQN [6].
- Experiment with Least-Squares in the Deep Deterministic Policy Gradients [5]

## 2.1 Dueling Deep Q-Learning - DuelingDQN

We present a recent modification to the vanilla DQN called Dueling DQN [9]. The core observation from the paper is that the Q-values $Q(s,a)$ can be divided into quantities: the value of the state $V(s)$ and the advantage of actions in this state, $A(s,a)$, where by definition $Q(s,a) = V(s) + A(s,a)$. In plain words, the advantage represents how much extra reward some particular action fro the state gives us. The contribution to DQNs is better training stability, faster convergence and better results on the Atari benchmark. The architecture of the network is as follows:



In order to make the mean value of the advantage of any state to be zero (to make $V(s)$ the expected value of the state), the output of the network is:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{N}\sum_k A(s,k)$$

We will now derive the LS-UPDATE equations for the Dueling DQN, as now we have two networks, and thus two approximators. The Q-function is of the form:

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{N}\sum_k A(s,k) \approx \phi_V^T(s,a)w_V + \phi_A^T(s,a)w_A - \frac{1}{N}\sum_k \phi_A(s,k)^T w_A$$

$$= \phi_V^T(s,a)w_V + \phi *_A^T (s,a)w_A$$

Where $\phi *_A = \phi_A - \frac{1}{N}\sum_k \phi_A(s,k)$.

We wish to find $w_A, w_V$ that minimize

$$||\phi_V^T(s,a)w_V + \phi *_A^T (s,a)w_A - y_i||_2$$

We get two equations with two variables with the solution:

$$w_V^{last} = (\phi_V^T\phi_V - \phi_V^T\phi_A(\phi_A^T\phi_A)^{-1}\phi_A^T\phi_V)^{-1}(\phi_V^T y_i - \phi_V^T\phi_A(\phi_A^T\phi_A)^{-1}\phi_A^T y_i)$$

$$w_A^{last} = (\phi_A^T\phi_A - \phi_A^T\phi_V(\phi_V^T\phi_V)^{-1}\phi_V^T\phi_A)^{-1}(\phi_A^T y_i - \phi_A^T\phi_V(\phi_V^T\phi_V)^{-1}\phi_V^T y_i)$$

With the added Bayesian regularization:

$$w_V^{last} = (\phi_V^T\phi_V - \phi_V^T\phi_A(\phi_A^T\phi_A + \lambda I)^{-1}\phi_A^T\phi_V + \lambda I)^{-1}(\phi_V^T y_i - \phi_V^T\phi_A(\phi_A^T\phi_A + \lambda I)^{-1}\phi_A^T y_i + \lambda w_{V_k}^{last})$$

$$w_A^{last} = (\phi_A^T\phi_A - \phi_A^T\phi_V(\phi_V^T\phi_V + \lambda I)^{-1}\phi_V^T\phi_A + \lambda I)^{-1}(\phi_A^T y_i - \phi_A^T\phi_V(\phi_V^T\phi_V + \lambda I)^{-1}\phi_V^T y_i + \lambda w_{A_k}^{last})$$

We denote:

$$\tilde{A_V} = \phi_V^T\phi_V - \phi_V^T\phi_A(\phi_A^T\phi_A + \lambda I)^{-1}\phi_A^T\phi_V + \lambda I$$
$$\tilde{b_V} = \phi_V^T y_i - \phi_V^T\phi_A(\phi_A^T\phi_A + \lambda I)^{-1}\phi_A^T y_i + \lambda w_{V_k}^{last}$$
$$\tilde{A_A} = \phi_A^T\phi_A - \phi_A^T\phi_V(\phi_V^T\phi_V + \lambda I)^{-1}\phi_V^T\phi_A + \lambda I$$
$$\tilde{b_A} = \phi_A^T y_i - \phi_A^T\phi_V(\phi_V^T\phi_V + \lambda I)^{-1}\phi_V^T y_i + \lambda w_{A_k}^{last}$$

Which yields:

$$w_V^{last} = \tilde{A_V}^{-1}\tilde{b_V}$$
$$w_A^{last} = \tilde{A_A}^{-1}\tilde{b_A}$$

In our work, we perform the LS-UPDATE step on the last hidden layer before the advantage layer and the value layer.

## 2.2 Boosted Fitted Q-Iteration - Boosted FQI

Boosted Fitted Q-Iteration (B-FQI) [6] is an Approximated Value Iteration offline algorithm that exploits a boosting procedure to estimate the action-value function. It essentially builds an approximation of the optimal action-value function by summing the approximators of the Bellman residuals across all iterations. We denote the following:

- The *empirical Bellman operator* $\hat{T}^*$:

$$(\hat{T}^*Q)(X_i, A_i) = R_i + \gamma max_{a'}Q(X_i^{'}, a^{'})$$

- The *empirical Bellman residual*:

$$\tilde{\varrho}_k = \hat{T}^*Q_k - Q_k$$

- The *truncation operator* $\beta_{B_k}$ such that for any function $f \in B(X \times A), \beta_B f(x,a) \in [-B, B]$ for some real $B > 0$

**Algorithm 1:** Boosted Fitted Q-Iteration

**Data:** $(D_n^{(i)})_{i=0}^K, (\beta_{B_i})_{i=0}^K, Q_0 = 0$
**for** $k = 0, , , K$ **do**
  $\hat{\varrho}_k \leftarrow \hat{T}^*Q_k - Q_k$ (w.r.t $(D_n^{(k)})$
  $Q_{k+1} \leftarrow Q_k + \beta_{B_k} argin f_{f\in F}||f - \hat{\varrho}_k||_{D_n^{(k)}}$
**end**
**return** $\overline{\pi}(x) = argmax_a Q_{K+1}(x,a), \forall x \in X$

In the linear approximation form, we get the following:

$$\tilde{A} = \frac{1}{N_{SRL}}\sum_{i=1}^{N_{SRL}}\left[\phi(s_i,a_i)^T\phi(s_i,a_i)\right]$$

$$\tilde{b} = \frac{1}{N_{SRL}}\sum_{i=1}^{N_{SRL}}\left[\phi(s_i,a_i)^T\beta_{B_k}\hat{\varrho}_k\right]$$

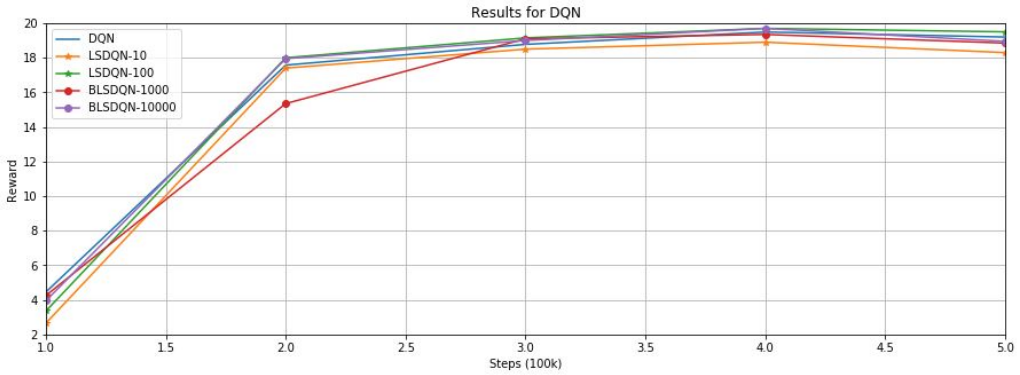



Figure 1: Results on vanilla-DQN (Pong)

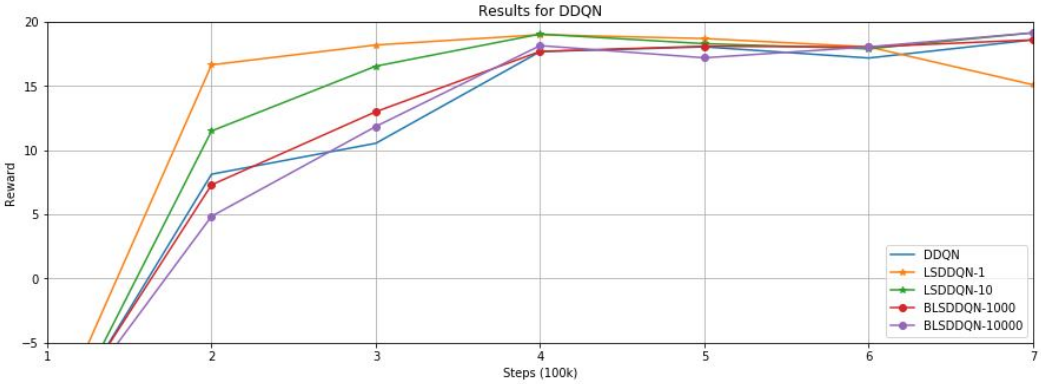



Figure 2: Results on Double DQN (Pong)

### 2.2.1 Boosted Deep Q-Network

*Boosted Deep Q-Network* [7] is a recent attempt at blending *Bellman residuals* with DRL architectures. This thesis shows that there is some little improvement in specific environments, but in the Atari environments, there is almost none, and sometimes even worse performance. Even though in our work we retrain using an SRL algorithm, we don't expect the boosted version to have better performance than the non-boosted one in the Atari environments. We wish to see that the boosted version does not degrade the performance, and can be used as an alternative.

## 2.3 FQI and B-FQI Algorithms with High Dimensional Observations

We perform the same procedure done in the original paper [10] on the game Pong for both FQI and B-FQI. That is, every 100,000 steps, we retrain the last hidden layer with an LS-UPDATE and evaluate the average reward over 20 roll-outs. For Pong, we used constant batch size of 64. The results are shown in figures 1, 2 and 3.

It can be seen that the results on the vanilla-DQN are not very impressive, while on the DDQN the performance is improved for all variations of LS-DQN. One should keep in mind that Pong is a fairly easy game that converges at an earlier step that other Atari games. The value of the regularization parameter $\lambda$ seems to have an impact on the performance, and it varies between FQI and B-FQI in magnitude. As for the Dueling architecture, the LS variation outperforms the original Dueling DQN, but not by much, at least for Pong.

From this analysis, it is clear that the question "when" to make an LS-UPDATE is important, and thus

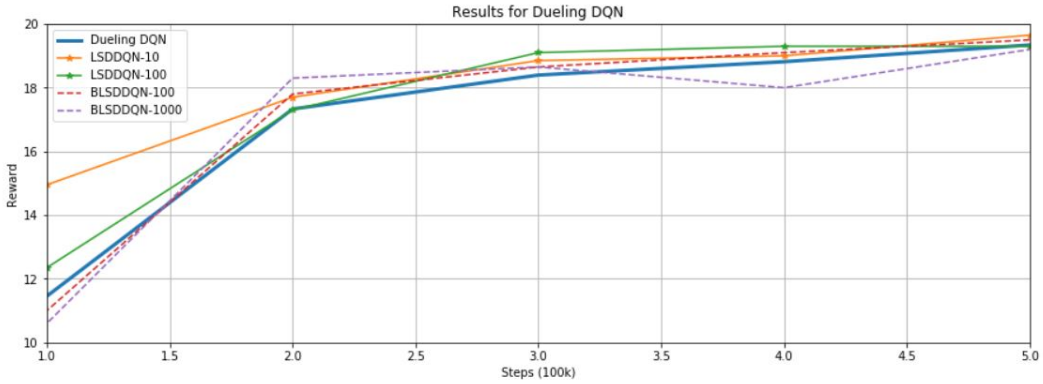



Figure 3: Results on Dueling DQN (Pong)

we suggest *Conditional LS-UPDATE*: performing a scheduled LS-UPDATE (e.g. every 100k frames), but use the LS weights only if the average reward on $M$ (e.g. $M = 20$) roll-outs is higher, otherwise, keeping the original weights. Also, it seems that it is worthwhile performing the updates at early stages, as in the last stages the performance is the same. This aligns with the conclusion from the original paper, where the final weights are very close to those from the LS-UPDATE.

## 2.4 Atari Experiments

Similarly to the original paper, we trained Atari games to test the algorithm's performance.

The first game we tested was Pong, using the following baseline: the base architecture is Dueling DQN, using batch size of 32 and a constant seed of 10. We used *conditional LS-UPDATE*, that is, the updated weights from LS-STEP are used only if they perform better on 20 roll-outs. The results are shown in figure 4. There were about 1.5 LS-UPDATES on average during both runs. It seems that both LS and Boosted LS (B-LS) performed better with different $\lambda$ values.

The second game we tested was Boxing, using the following baseline: the base architecture is Dueling DQN, using batch size of 32 and a constant seed of 2019. Due to time limits, we chose to use Boosting with $\lambda = 1000$ based on the results from the previous section on Pong. We ran 2 runs with the same parameters and took the mean. The reason we do this is because even when we use constant seed, it is near impossible to control the CUDA-NN seed when training on the GPU. Moreover, we used *conditional LS-UPDATE* as described before. The results are shown in figure 5. There were about 5 LS-UPDATES during both runs. It seems that the LS version performed a little better, and then lost it towards the end. We assume that even though we kept a low $\lambda$ to avoid drastic changes to the weights, as the training reaches optimal values, even the slightest change can harm the fine tuning process.

## 2.5 Least-Squares Deterministic Policy Gradients

Deep deterministic policy gradients (DDPG) [5] is a variation of the A2C method but unlike A2C, is an *off-policy* algorithm. In A2C, the actor estimates the stochastic policy, which returns the probability distribution over discrete actions. The action taken is sampled from this distribution. In DDPG the policy is deterministic, that is, provides us with the action to take from the current state. In this way, it is possible to apply the chain rule to the Q-value, and by maximizing the Q, the policy will be improved as well. In the continuous domain (e.g. robotic environments) it works as follows: the actor network takes in the current state and outputs the action to take. The actor NN return N values, one for every action (e.g., if the robot has 4 motors, the output is the action to take for every motor). The critic job's is to estimate the Q-value (the discounted reward of the action taken in some state). As the action is a tensor of numbers, the critic now accepts two inputs: the state and the action. The output is a single number (the Q-value). Denoting the actor function as $\mu(s)$ yields the critic function $Q(s, \mu(s))$.

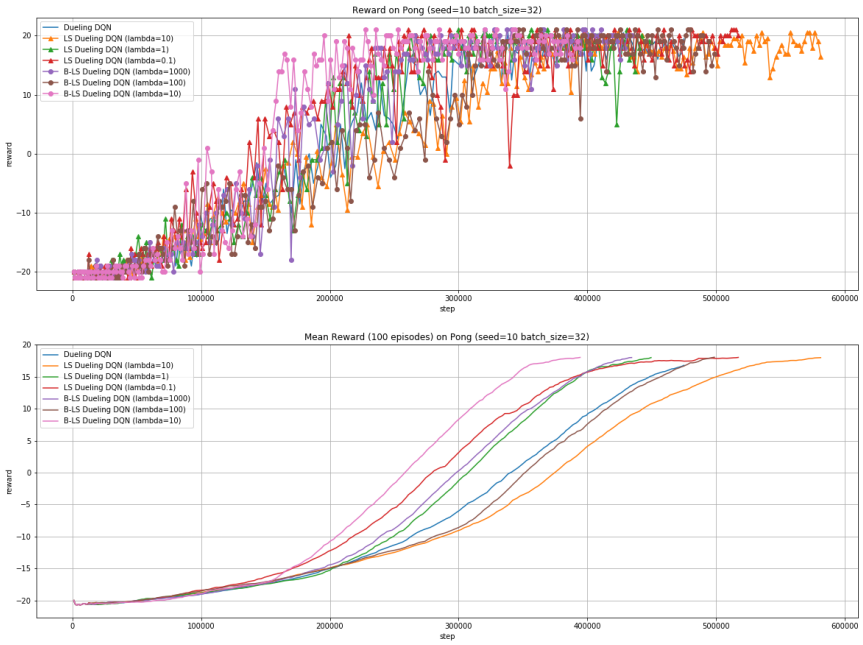



Figure 4: Results on Dueling DQN (Pong)

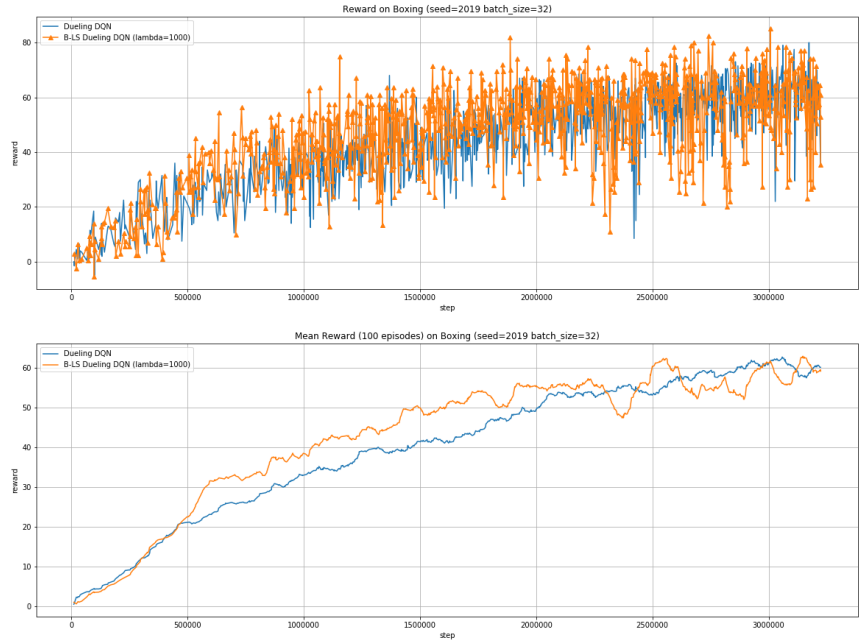



Figure 5: Results on Dueling DQN (Boxing)

| | |
|---|---|
| DDPG | 3.885 |
| B-LS-DDPG ($\lambda = 100$) | 4.111 |
| LS-DDPG ($\lambda = 1$) | 4.428 |

Figure 6: Best Reward in MinitaurBulletEnv-v0 in 4M steps

| | |
|---|---|
| DDPG | 309.266 |
| B-LS-DDPG ($\lambda = 10000$) | 316.064 |
| LS-DDPG ($\lambda = 100$) | 307.782 |

Figure 7: Best Reward in BipedalWalker-v2 in 3.5M steps

The output of the critic is the target function we wish to maximize. At every step of the optimization, we wish to to modify the actor's weights to improve the total reward, that is, use the gradient of the policy. There are two optimization steps, the first is for the critic $Q(s, \mu(s))$ and the second is achieved by the chain rule: $\nabla_a Q(s, a) \nabla_{\theta_\mu} \mu(s)$. Exploiting the fact that this is an *off-policy* algorithm, we can use a replay buffer, thus allowing us to apply LS-UPDATE on the critic network before updating the actor network. DDPG, being a deterministic algorithm, has an issue of exploration. This is solved by adding random noise to the actions using *OU processes*: $x_{t+1} = x_t + \theta(\mu - x_t) + \sigma N$ where $x$ is the process's state, $\mu$ is the current action value, $\theta, \sigma$ are hyper-parameters and $N$ is a random normal noise. The DDPG scheme is displayed in figure 8. In our experiments we hope to exploit the big batch updates to improve the critic value, and thus improving the actor.

#### 2.5.1 Results on PyBullet's MinitaurBulletEnv-v0 Environment

This is a robotic simulator of a 4-motor robot, where the target is to walk as long as possible. The robot is depicted in figure 8. The best rewards achieved in 4M steps are shown in figure 6. We used batch size of 64 and perform a LS-STEP every 100k steps. It can be seen that LS-DDPG with $\lambda = 1$ performed the best.

#### 2.5.2 Results on OpenAI's BipedalWalker-v2 Environment

This is a 2D robotic simulator, where the target is to walk as far and as fast as possible. The robot is also depicted in figure 8. Unlike the Atari environments, we don't use CNNs and the merit of using the GPU instead of the CPU is only slightly better, and thus we use the CPU in order to be able to make the seed absolutely constant. We run the vanilla DDPG and test some variations of LS-DDPG. We use batch size of 64, seed of 2019, LS-STEP every 100k steps and limit the runs to 3.5M steps. We also implemented noise decay for the OU process in order to make the training more stabilized (for both DDPG and LS-DDPG). The training process is shown in figure 7, where we plot some of the $\lambda$ values we tried. It worth noting that because we trained on the CPU, the mean reward for every run is the same until the point of the first LS-STEP (about 110k steps). The main surprise was the early boost all of the LS-DDPG versions have given the training process, which accelerated reaching the best reward. The best rewards achieved in 3.5M steps are shown in figure 7. It can be seen that B-LS-DDPG with $\lambda = 10000$ performed the best and was superior in every term.

### 2.6 Future Work

- Given more resources, BLS-DQN should be examined on the games in the original paper and other, more difficult, games.
- LS-DDPG performance should be explained with more theoretical basis.
- Least-Squares update can be implemented in other algorithms such as D4PG.

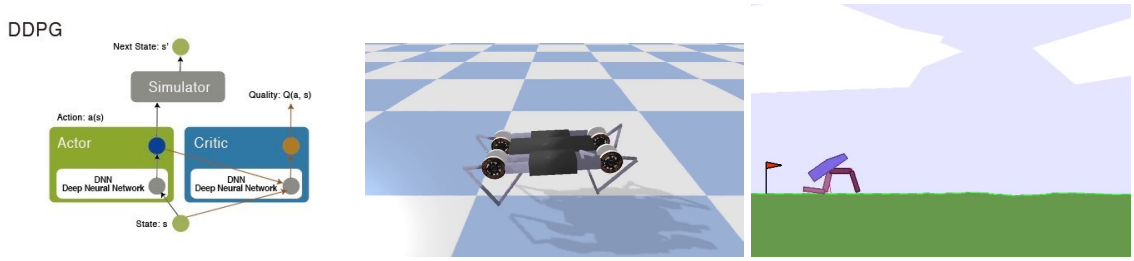



Figure 8: DDPG Scheme and Environments (MinitaurBulletEnv-v0 and BipedalWalker-v2)

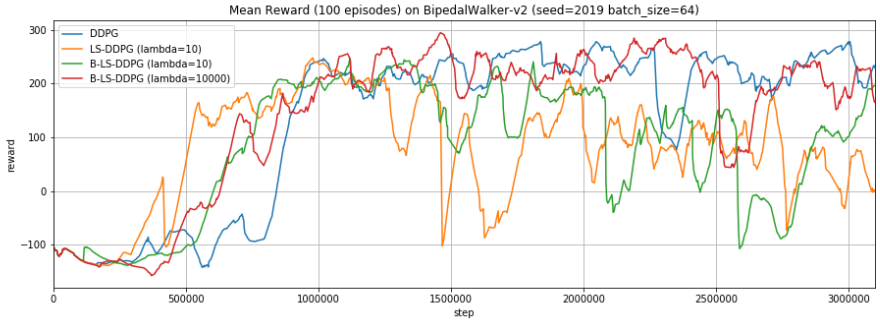



Figure 9: Results on LS-DDPG (BipedalWalker-v2)

- $\lambda$ -*scheduling* - as we have seen, the NN's are very sensitive to the LS-UPDATES. One should try and increase $\lambda$ as the training reaches near-optimal values, so the changes to weights become less drastic.

### 2.7 Appendix

GitHub:

- LS-DQN - https://github.com/taldatech/pytorch-ls-dqn
- LS-DDPG - https://github.com/taldatech/pytorch-ls-ddpg

Video:

https://youtu.be/i8Cnas7QrMc

## References

[1] Kamyar Azizzadenesheli, Emma Brunskill, and Animashree Anandkumar. Efficient exploration through bayesian deep q-networks. *arXiv:1802.04412*, 2018.

[2] Jeff Donahue, Yangqing Jia, Oriol Vinyals, Judy Hoffman, Ning Zhang, Eric Tzeng, and Trevor Darrell. Decaf: A deep convolutional activation feature for generic visual recognition. *Proceedings of the 31st International Conference on Machine Learning, PMLR 32(1):647-655, 2014*, 2014.

[3] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Deep q-learning from demonstrations. *AAAI 2018*, 2018.

[4] J. Zico Kolter and Andrew Y. Ng. Regularization and feature selection in least-squares temporal difference learning. *ICML 2009*, 2009.

[5] 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, 2015.

[6] Samuele Tosatto, Matteo Pirotta, Carlo DEramo, and Marcello Restelli. Boosted fitted q-iteration. *Proceedings of the 34th International Conference on Machine Learning, PMLR 70:3434-3443, 2017*, 2017.

[7] J. Tschirner. *Boosted Deep Q-Network*. PhD thesis, Techbische Universitat Darmstadt, 2018.

[8] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *ASSOCIATION FOR THE ADVANCEMENT OF ARTIFICIAL INTELLIGENCE, AAAI 2016*, 2016.

[9] Ziyu Wang, Tom Schaulm, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. *ICML'16 Proceedings of the 33rd International Conference on International Conference on Machine Learning*, 2016.

[10] Tom Zhavy, Nir Levine, Daniel Mankowitz, Aviv Tamar, and Shie Mannor. Shallow updates for deep reinforcement learning. *Neural Information Processing Systems (NIPS) 2017*, 2017.