Optimizing Q-learning with K-FAC Algorithm

Roman Beltiukov[1] [0000−0001−8270−0219]

Peter the Great St. Petersburg Polytechnic University
Saint-Petersburg, Russia
maybe.hello.world@gmail.com

Abstract. In this work, we present intermediate results of the application of Kronecker-factored Approximate curvature (K-FAC) algorithm to the Q-learning problem. Being more expensive to compute than plain stochastic gradient descent, K-FAC allows the agent to converge a bit faster in terms of epochs compared to Adam on simple reinforcement learning tasks and tend to be more stable and less strict to hyperparameters selection. Considering the latest results, we show that DDQN with K-FAC learns more quickly than with other optimizers and continuously improves in contradiction to similar to Adam or RMSProp.

Keywords: Q-learning · K-FAC · Reinforcement learning · Natural gradient

1 Introduction

During the last years, many successes in the field of reinforcement learning were achieved by using Deep Q-network (DQN) algorithms. Starting with [1], where authors combined Q-learning with the latest convolutional neural networks, a rather large amount of articles were published and proposed both architecture and algorithm changes to achieve higher scores and stability of Q-learning [2,3,4].

Being a Q-function optimizing algorithm compared to policy-optimization algorithms, Q-learning is subject to instability, leading to the unpredictable degradation of an agent. The majority of articles mentioned above stabilized the learning process and allowed to reach higher results, but the problem is still relevant.

According to [5], using natural gradient descent can improve the speed of convergence process measured by a number of steps by increasing the time of calculation of each step. There are only few articles about applying second-order optimization methods to Deep Q-networks algorithms, which either use heavy computational realizations of Fisher matrix calculation, leading to significantly increased training time [6] or suppose serious neural network architecture changes for natural gradient descent realization [7]. Meanwhile, in the area of policy-gradient algorithms, second-order optimization techniques become more and more popular and widely used [8,9]. These reasons, together with a request from OpenAI to research this topic, led to ideas for further work.

In this article, we attempted to apply second-order optimization techniques like natural gradient descent to the Q-learning in order to improve the stability of learning and decrease training time.

2 Background

2.1 Reinforcement Learning

We consider the standard RL framework [10] where an agent is involved in Markov Decision Process (MDP) with infinite horizon. An MDP is defined by the tuple (S, A, R, P, γ), which consists of a set of states S, a set of actions A, a reward function $R(s, a)$, transition probability function $P(s, a)$ and discount factor γ. At step t the agent observes state $s_t \in S$, selects some action $a_t \in A$ and then receives scalar reward r and next state s_{t+1} given by transition function. The agent aims to maximize γ-discounted cumulative reward by finding optimal policy π_θ.

Value-based methods optimize policies by optimizing the corresponding value function $Q^\pi(s, a)$, which estimates expected future reward that can be obtained by the agent that takes action a in given state s and then acts according to the policy π. The optimal value function $Q^*(s, a)$, in this case, provides the most correct prediction of reward and is determined by solving the Bellman equation:

$$Q^*(s, a) = E\left[R(s, a) + \gamma \max_{a'} E\left[Q^*(s', a')\right]\right] \tag{1}$$

The optimal policy π^* then is defined as $\pi^*(s) = argmax_{a \in A} Q^*(s, a)$.

2.2 Deep Q-networks

Presented in [1], DQN approach approximates the Q-function with a deep neural network that outputs values of all possible actions for a given observation. Usually, during training, DQN agents use such techniques as replay buffer to store states, actions, rewards, and transitions to learn and separate target networks for stabilizing the learning process.

The double deep Q-learning (DDQN) method [2] proposes using the current network for calculating argmax over next state values with different loss function:

$$J = R(s, a) + \gamma Q(s_{t+1}, argmax Q^t(s_{t+1}, a_{t+1})) \tag{2}$$

Using such an approach leads to better performance and reducing overestimations of action values.

2.3 Kronecker-factored Approximate Curvature

Natural gradient descent [5] (NGD) is a second-order optimization method that uses a Fisher information matrix to take steepest descent direction in model

distribution space instead of parameter space. Using NGD usually allows converging in a fewer number of iterations than with stochastic gradient descent methods.

A recently proposed Kronecker-factored approximate curvature algorithm [11] (K-FAC) uses a computationally efficient approximation of the Fisher matrix through Kronecker factorization to perform natural gradient updates. By increasing computing time of single step approximately by 30% for full-connected layers, K-FAC allows to converge faster in terms of a number of iterations then SGD with momentum.

3 Method

For this research, Tensorflow open-source implementation of the K-FAC algorithm was used [12]. For optimizer testing, OpenAI Baselines' [13] DDQN implementation was adopted, and necessary changes for K-FAC optimizer stabilization were also made. Two network architectures were used for evaluating optimizer:

1. For discrete environments from "classic control environments list" of OpenAI Gym[14], and also for LunarLander-v2, 3-layer fully connected neural network was used. The first and second layers have 64 and 16 neurons with hyperbolic tangent activation; the last layer has as many neurons as a number of available actions in the corresponding environment.
2. For Atari environments from OpenAI Gym, a network, similar to [1], was used.

A list of current changes mainly includes merging K-FAC implementation code with both networks, changing some hyperparameters of networks, etc. The current implementation is a work-in-progress and subject to change in the future.

Experiments were held on GPU partition with Nvidia Tesla K40. Full code and preprocessing settings for each environment and also current results are available in projects' repository[1].

4 Current Status and Intermediate Results

By now, the implementation of DDQN with K-FAC optimizator has been completed, and intermediate results were obtained for some classic control environments of the OpenAI Gym framework. For each optimizer, optimal hyperparameters were selected by a grid search with averaging across 20 different seeds. In particular, in *CartPole-v1* K-FAC-DDQN realization showed the faster speed of convergence on optimal hyperparameters than similar DDQN realization with *Adam* [15] or *RMSProp* [16]. Corresponding results are shown in Fig. 1. Spending about 30% more time for each iteration, K-FAC-DDQN implementation requires a fewer number of iterations for reaching a similar reward. Additional results are shown in Table 1 and Fig. 2. For some environments (like *MountainCar-v0*),

[1] https://github.com/maybe-hello-world/qfac

memory ability of network is rather low, and plateau of reward plot is not close to state-of-the-art examples.

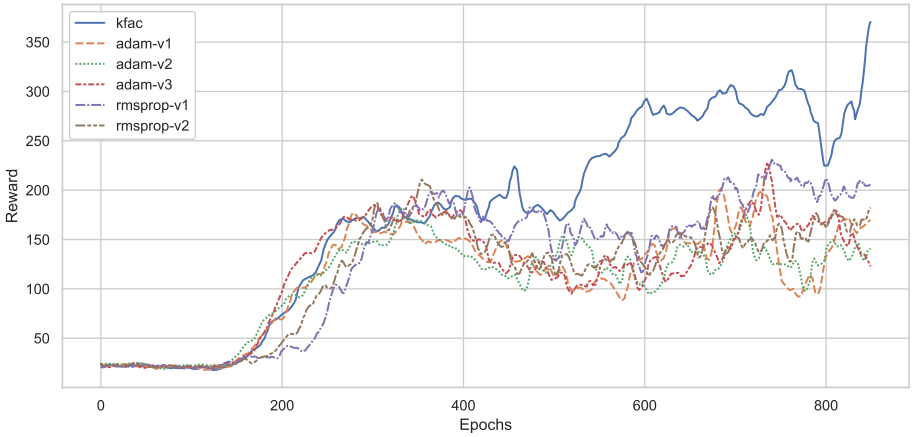


Fig. 1: Mean performance of optimizers with optimal hyperparameters across 20 random seeds on *CartPole-v1*.

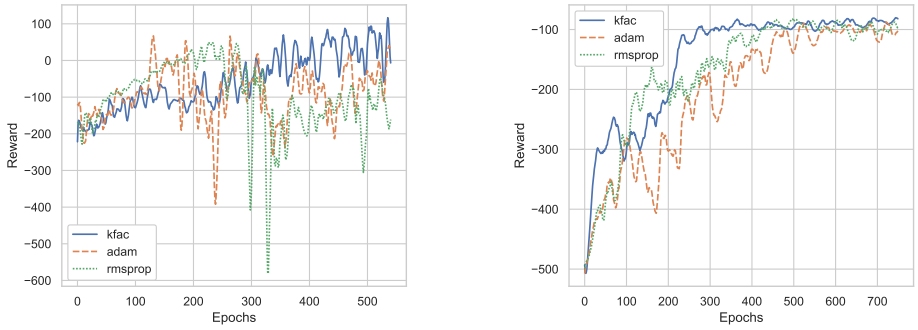
Table 1: The average total reward among the last 100 games for various optimizers by running an ϵ-greedy policy with $\epsilon = 0.1$ for a fixed number of steps.

Optimizer	MountainCar-v0	CartPole-v1	Acrobot-v1	LunarLander-v2
Adam	-122	183	-108	-46
RMSProp	-119	194	-97	-136
K-FAC	-124	**321**	-92	**38**

During the training process, another interesting observation was made – K-FAC implementation requires less attention to hyperparameters of optimizator (except dumping) than *Adam* or *RMSProp*. On average, the K-FAC-DDQN implementation obtained more stable and higher results during the grid search process over the hyperspace of hyperparameters than other optimizers.

Also sometimes [17,18] while working with graphical input data, researchers use byte interpretation with integer range [0..255] without scaling to [0..1] float range. Unfortunately, together with K-FAC optimizer, this leads to instability of Cholesky transformation and fails to compute, resulting in obligatory scaling of input data for the network.

Summarizing, K-FAC optimizer tends to be more smoothly growing and avoiding sudden loss of performance during training. Possibly for very simple tasks, other optimizers could be much more efficient hopping up to optimum,



(a) LunarLander-v2 (b) Acrobot-v1

Fig. 2: Mean performance of optimizers with optimal hyperparameters in different environments. On average K-FAC mainly do not allow burst improvements but also tends to avoid performance degradation.

but for more hard environments, K-FAC should be considered as an option. Although, it's important to remember that using second-order optimization algorithms requires more memory for Fisher matrix computation and additional computational time for the algorithm. Amount of memory and computational time depends on network layers (from about 30% for fully-connected layers and up to 2-3 times for convolution layers) and input data so in case of training heavy convolutional networks using K-FAC could be impossible or less effective, if use certain K-FAC options.

5 Discussion

At present, the hypothesis that using the K-FAC an RL agent can achieve a more stable and predictable learning process is neither proven nor disproved – this requires the searching for optimal hyperparameters and numerous experiments in several Gym environments. Nevertheless, intermediate results allow us to speak about the possibility of using the K-FAC algorithm for Q-learning agents. In the case of stabilizing the learning process by K-FAC, it could be possible to achieve higher total rewards or train with less number of iterations.

At this moment, only DQN and DDQN algorithms were implemented due to the difficulty of adapting the K-FAC method to newer algorithms such as Rainbow DQN [19]. However, using a less stable implementation of the DQN agent may lead to a more noticeable impact of K-FAC on the result.

Further, we plan to adapt the K-FAC method to later DQN improved implementations, like Rainbow DQN, as well as conducting numerous experiments with the existing implementation in various OpenAI Gym environments.

Acknowledgments

We would like to thank Olga Tushkanova for helpful comments, constructive criticism, and useful feedback. The results of the work were obtained using the computational resources of Peter the Great Saint-Petersburg Polytechnic University Supercomputing Center (www.scc.spbstu.ru). The research was partially funded by the 5-100-2020 program and SPbPU university.

References

1. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing Atari with Deep Reinforcement Learning (dec 2013), http://arxiv.org/abs/1312.5602
2. van Hasselt, H., Guez, A., Silver, D.: Deep Reinforcement Learning with Double Q-learning (sep 2015), http://arxiv.org/abs/1509.06461
3. Bellemare, M.G., Dabney, W., Munos, R.: A Distributional Perspective on Reinforcement Learning (jul 2017), http://arxiv.org/abs/1707.06887
4. Fortunato, M., Azar, M.G., Piot, B., Menick, J., Osband, I., Graves, A., Mnih, V., Munos, R., Hassabis, D., Pietquin, O., Blundell, C., Legg, S.: Noisy Networks for Exploration (jun 2017), http://arxiv.org/abs/1706.10295
5. Amari, S.i.: Natural Gradient Works Efficiently in Learning. Neural Computation **10**(2), 251–276 (feb 1998). https://doi.org/10.1162/089976698300017746, http://www.mitpressjournals.org/doi/10.1162/089976698300017746
6. Knight, E., Lerner, O.: Natural Gradient Deep Q-learning (mar 2018), http://arxiv.org/abs/1803.07482
7. Desjardins, G., Simonyan, K., Pascanu, R., Kavukcuoglu, K.: Natural Neural Networks (jul 2015), http://arxiv.org/abs/1507.00210
8. Schulman, J., Levine, S., Moritz, P., Jordan, M.I., Abbeel, P.: Trust Region Policy Optimization (feb 2015), https://arxiv.org/abs/1502.05477
9. Wu, Y., Mansimov, E., Liao, S., Grosse, R., Ba, J.: Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation (aug 2017), http://arxiv.org/abs/1708.05144
10. Sutton, R.S., Barto, A.G., et al.: Introduction to reinforcement learning, vol. 135. MIT press Cambridge (1998)
11. Martens, J., Grosse, R.: Optimizing Neural Networks with Kronecker-factored Approximate Curvature (mar 2015), http://arxiv.org/abs/1503.05671
12. Martens, J., Tankasali, V., Duckworth, D., Johnson, M., Zhang, G., Koonce, B.: tensorflow\kfac. https://github.com/tensorflow/kfac, [Online; accessed 25-March-2019]
13. Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., Zhokhov, P.: Openai baselines. https://github.com/openai/baselines (2017)
14. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: OpenAI Gym (jun 2016), http://arxiv.org/abs/1606.01540
15. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization (dec 2014), http://arxiv.org/abs/1412.6980
16. Hinton, G.: RMSProp. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, [Online; accessed 25-March-2019]

17. Ecoffet, A., Huizinga, J., Lehman, J., Stanley, K.O., Clune, J.: Go-Explore: a New Approach for Hard-Exploration Problems (jan 2019), http://arxiv.org/abs/1901.10995
18. Resnick, C., Raileanu, R., Kapoor, S., Peysakhovich, A., Cho, K., Bruna, J.: Backplay: "Man muss immer umkehren" (jul 2018), http://arxiv.org/abs/1807.06919
19. Hessel, M., Modayil, J., van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., Silver, D.: Rainbow: Combining Improvements in Deep Reinforcement Learning (oct 2017), http://arxiv.org/abs/1710.02298