Natural Gradient Deep Q-learning

Ethan Knight*
ethan.h.knight@gmail.com
Stanford Cognitive & Systems Neuroscience Lab
The Nueva School

Osher Lerner
osherler@gmail.com
The Nueva School

Abstract

This paper presents findings for training a Q-learning reinforcement learning agent using natural gradient techniques. We compare the original deep Q-network (DQN) algorithm to its natural gradient counterpart (NGDQN). Measuring NGDQN and DQN performance on classic controls environments without target networks. We find that NGDQN performs favorably relative to DQN, converging to significantly better policies faster and more frequently. These results indicate that natural gradient could be used for value function optimization in reinforcement learning to accelerate and stabilize training.

1 Introduction

Natural gradient was originally proposed by Amari as a method to accelerate gradient descent [1998]. Rather than exclusively using the loss gradient or including second-order (curvature) information, natural gradient uses the "information" found in the parameter space of the model.

Natural gradient has been successfully applied to several deep learning domains and has been used to accelerate the training of reinforcement learning systems [Desjardins et al., 2015, Kakade, 2001, Schulman et al., 2015, Wu et al., 2017]. To motivate our approach, we hoped that using natural gradient would accelerate the training of a DQN as it did with other reinforcement learning systems, making our system more sample-efficient, thereby addressing one of the major problems in reinforcement learning. We also hoped that since natural gradient stabilizes training (e.g. natural gradient is relatively unchanged when changing the order of training inputs [Pascanu and Bengio, 2013]), NGDQN could be able to achieve good results without a target network, and converge to good solutions with more stability.

In our experiments, we observed both effects. When training without a target network, NGDQN converged much faster and more frequently than our DQN baseline, and the NGDQN training appeared much more stable.

This paper was inspired by the Requests for Research list published by OpenAI, which has listed the application of natural gradient techniques to Q-learning since June 2016 [2016, 2018]. This paper presents the first successful attempt to our knowledge: our method to accelerate the training of Q-networks using natural gradient.

2 Background

In reinforcement learning, an agent is trained to interact with an environment to maximize cumulative reward. An agent interacts with an environment by observing a state s, performing an action a, and receiving a new state s' and reward r'. Often, this environment is modeled as a Markov Decision Process (MDP), which defines a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$, R, the expected reward given s

*Contributor statement and acknowledgements at end of paper

and a, dynamics P which give the probability of a state s' given prior state s and action a [Sutton and Barto, 1998]. The discount factor γ is also defined, which specifies how the cumulative reward R_t is calculated [Mnih et al., 2013]:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{1}$$

The agent attempts to learn a policy π to maximize this cumulative reward.

2.1 Q-learning

Q-learning [Watkins, 1989, Rummery and Niranjan, 1994] is a model-free reinforcement learning algorithm which works by gradually learning $Q(s, a)$, the expectation of the cumulative reward. The Bellman equation defines the optimal Q-value Q^* [Sutton and Barto, 1998, Hester et al., 2017]:

$$Q^*(s, a) = \mathbb{E}\left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q^*(s, a)\right] \tag{2}$$

This function Q can be then optimized through value iteration, which defines the update rule $Q(s, a) \leftarrow \mathbb{E}\left[r + \gamma \max_{a'} Q(s', a')|s, a\right]$ [Sutton and Barto, 1998, Mnih et al., 2013]. Additionally, the optimal policy π is defined as $\pi(s) = \text{argmax}_{a \in \mathcal{A}} Q^*(s, a)$ [Sutton and Barto, 1998, Hester et al., 2017].

To train a Q-learning agent, we often use an ϵ-greedy policy. Initially, the training agent acts nearly randomly in order to explore potentially successful strategies. As the agent learns, it acts randomly less (this is sometimes called the "exploit" stage, as opposed to the prior "explore" stage). Mathematically, the probability of choosing a random action ϵ is gradually linearly annealed over the course of training.

Q-learning was recently used to play Atari games using convolutional networks by Mnih et al. [2013]. For a state s, action a, and learning rate α, the function Q estimates the future reward and is updated as follows:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha y \tag{3}$$

If the agent has not yet reached the end of the task, given discount factor γ and reward r, the target is defined as

$$y = r + \gamma \max_{a'} Q(s', a') \tag{4}$$

In environments with defined endings, the final timestep is defined as $y = r$ since there is no future reward. In deep Q-learning, this mapping is learned by a neural network.

A neural network can be described as a parametric function approximator that uses "layers" of units, each containing weights, biases and activation functions, called "neurons". Each layer's output is fed into the next layer, and the loss is backpropagated to each layer's weights in order to adjust the parameters according to their effect on the loss.

For deep Q-learning, the neural network, parameterized by θ, takes in a state s and outputs a predicted future reward for each possible action a_i. The loss of this network is defined as follows:

$$\mathcal{L} = \mathbb{E}\left[(y - Q(s, a; \theta))^2\right] \tag{5}$$

where $Q(s, a; \theta)$ is the output of the network corresponding to action a, and

$$y = \mathbb{E}_{s' \sim \varepsilon}\left[r + \gamma \max_{a'} Q(s', a'; \theta)_j \middle| s, a_i\right] \tag{6}$$

Notice that we take the mean-squared-error between the expected Q-value and actual Q-value. The neural network is optimized over the course of numerous iterations through some form of gradient descent. In the original DQN (deep Q-network) paper, an adaptive gradient method is used to train this network [Mnih et al., 2013].

Deep Q-networks use experience replay to train the Q-value estimator on a randomly sampled batch of previous experiences (essentially replaying past remembered events back into the neural network) [Lin, 1992]. Experience replay makes the training samples independent and identically distributed (i.i.d.), unlike the highly correlated consecutive samples which are encountered during interaction with the environment [Schaul et al., 2015]. This is a prerequisite for many SGD convergence theorems. Additionally, we use an ϵ-greedy policy in both our DQN baseline and in NGDQN.

We combine these two approaches, using natural gradient to optimize an arbitrary neural network in Q-learning architectures.

3 Natural gradient for Q-learning

Gradient descent optimizes parameters of a model with respect to a loss function by "descending" down the loss manifold. To do this, we take the gradient of the loss with respect to the parameters, then move in the opposite direction of that gradient [Goodfellow et al., 2016]. Mathematically, gradient descent proposes the point given a learning rate of α and parameters θ: $\theta - \alpha \nabla_\theta \mathcal{L}(x, y; \theta)$.

A commonly used variant of gradient descent is stochastic gradient descent (SGD). Instead of calculating the entire gradient at a time, SGD uses a mini-batch of training samples: $\theta - \alpha \nabla_\theta \mathcal{L}(x_i, y_i; \theta)$. Our baselines use Adam, an adaptive gradient optimizer, which is a modification of SGD [Kingma and Ba, 2014].

However, this approach of gradient descent has a number of issues. For one, gradient descent will often become very slow in plateaus where the magnitude of the gradient is close to zero. Also, while gradient descent takes uniform steps in the parameter space, this does not necessarily correspond to uniform steps in the output distribution. Natural gradient attempts to fix these issues by incorporating the inverse Fisher information matrix, a concept from statistical learning theory [Amari, 1998].

Essentially, the core problem is that Euclidean distances in the parameter space do not give enough information about the distances between the corresponding outputs, as there is not a strong enough relationship between the two [Foti, 2013]. Kullback and Leibler define a more expressive distribution-wise measure, as follows [1951].

$$KL(\mu_1|\mu_2) = \int_{-\infty}^{\infty} \mu_1(s) \log \frac{\mu_1(s)}{\mu_2(s)} \, \mathrm{d}s \tag{7}$$

However, since $KL(\mu_1|\mu_2) \neq KL(\mu_1|\mu_2)$, symmetric KL divergence, also known as Jensen-Shannon (JS) divergence, is defined as follows [Foti, 2013]:

$$KL(\mu_1|\mu_2) := \frac{1}{2} \left(KL(\mu_1|\mu_2) + KL(\mu_2|\mu_1) \right) \tag{8}$$

To perform gradient descent on the manifold of functions given by our model, we use the Fisher information metric on a Riemannian manifold. Since symmetric KL divergence behaves like a distance measure in infinitesimal form, a Riemannian metric is derived as the Hessian of the divergence of symmetric KL divergence [Pascanu and Bengio, 2013]. We give Pascanu and Bengio's definition, which assume that the probability of a point sampled from the network is a gaussian with the network's output as the mean and with a fixed variance. Given some probability density function p, input vector s, and parameters θ [Pascanu and Bengio, 2013]:

$$\mathbf{F}_\theta = \mathbb{E}_{s,q}[(\nabla \log p_\theta(q|s))^T (\nabla \log p_\theta(q|s))] \tag{9}$$

Finally, to achieve uniform steps on the output distribution, we use Pascanu and Bengio's derivation of natural gradient given a loss function $\mathcal{L}$ [2013]:

$$\nabla \mathcal{L}_N = \nabla \mathcal{L} {\mathbf{F}_\theta}^{-1} \tag{10}$$

Taking the second-order Taylor expansion [Pascanu and Bengio, 2013] gets:

$$KL(p_\theta|p_{\theta+\Delta\theta}) \approx \frac{1}{2} \Delta\theta^T \mathbf{F}_\theta \Delta\theta \tag{11}$$

Using this approximation and solving the Lagrange multiplier for minimizing the loss of parameters updated by $\Delta\theta$ (approximated by a first order Taylor series) under a the constraint of a constant symmetric KL distance, one can derive the following equations for the information matrices.

As the output probability distribution is dependent on the final layer activation, Pascanu and Bengio [2013] give the following derivation for a layer with a linear activation (interpreted as a conditional Gaussian distribution), here adapted for Q-learning, where β is defined as the standard deviation:

$$p_\theta(q|s) = \mathcal{N}(q|Q(s,\theta),\beta^2) \tag{12}$$

In this formulation, since the information is only dependent on the final layer's activation we can use different activations in the hidden layers without changing the Fisher information. As in Pascanu and Bengio [2013], the Fisher information can be derived where $\mathbf{J}_Q$ corresponds to the Jacobian of the output vector as follows:

$$\mathbf{F}_{\text{linear}} = \beta^2 \mathbb{E}_{s\sim d^\pi(s)}\left[\mathbf{J}_Q^T \mathbf{J}_Q\right] \tag{13}$$

4 Related work

We borrow heavily from the approach of Pascanu and Bengio [2013], using their natural gradient for deep neural networks formalization and implementation in our method.

Next, we look at work on a different method of natural gradient descent by Desjardins et al. [2015]. In this paper, algorithm called "Projected Natural Gradient Descent" (PRONG) is proposed, which also considers the Fisher information matrix in its derivation. While our paper does not explore this approach, it could be an area of future research, as PRONG is shown to converge better on multiple data-sets, such as CIFAR-10 [Desjardins et al., 2015].

Additional methods of applying natural gradient to reinforcement learning using reinforcement learning algorithms such as policy gradient and actor-critic are explored in Kakade [2001] and Peters et al. [2005]. In both works, the natural variants of their respective algorithms are shown to perform favorably compared to their non-natural counterparts. Details on theory, implementation, and results are in their respective papers.

Insights into the mathematics of optimization using natural conjugate gradient techniques are provided in the work of Honkela et al. [2015]. These methods allow for more efficient optimization in high dimensions and nonlinear contexts.

The Natural Temporal Difference Learning algorithm applies natural gradient to reinforcement learning systems based on the Bellman error, although Q-learning is not explored [Tesauro, 1995]. The authors use natural gradient with residual gradient, which minimizes the MSE of the Bellman error and apply natural gradient to SARSA, an on-policy learning algorithm. Empirical experiments show that natural gradient again outperforms standard methods in the tested environments.

Finally, to our knowledge, the only one other published or publicly available version of natural Q-learning was created by [Barron et al., 2016]. In this work, the authors re-implemented PRONG and verified its effectiveness at MNIST. However, when the authors attempted to apply it to Q-learning they got negative results, with no change on CartPole and worse results on GridWorld.

5 Methods

In our experiments, we use a standard method of Q-learning to act on the environment. Lasagne [Dieleman et al., 2015], Theano [Theano Development Team, 2016], and AgentNet [Yandex, 2016] complete the brunt of the computational work. Because our implementation of natural gradient modified from Pascanu and Bengio [2013] fits an X to a mapping y, rather than directly back-propagating a loss the model uses a target value change similar to that described in 5. We also decay the learning rate by multiplying it by a constant factor every iteration.

As the output layer of our Q-network has a linear activation function, we use the parameterization of the Fisher information matrix for linear activations, which determines the natural gradient. For this, we refer to equation 13, approximated at every batch.

The MinRes-QLP Krylov Subspace Descent Algorithm [Choi et al., 2011] calculates the change in parameters according to the Fisher Information Matrix as in Pascanu and Bengio [2013] by efficiently

solving the system of linear equations relating the desired change in parameters to the gradients of the loss (see Algorithm 1). Our implementation runs on the OpenAI Gym platform which provides several classic control environments, such as the ones shown here, as well as other environments such as Atari [Brockman et al., 2016]. The current algorithm takes a continuous space and maps it to a discrete set of actions.

In Algorithm 1, we adapt Mnih et al.'s Algorithm 1 and Pascanu and Bengio's Algorithm 2 [2013, 2013]. Because these environments do not require preprocessing, we have omitted the preprocessing step, however this can easily be re-added. In our experiments, Δ_α was chosen somewhat arbitrarily to be $1 - 7e^{-5}$, and α was selected according to our grid-search (see: Hyperparameters). According to our grid search, we either leave the damping value unchanged or adjust it according to the Levenberg-Marquardt heuristic as used in Pascanu and Bengio [2013] and Martens [2010].

Algorithm 1 Natural Gradient Deep Q-Learning with Experience Replay

Require: Initial learning Rate α_0
Require: Learning rate decay $\Delta\alpha$
Require: Function `update_damping`
  Initialize replay memory $\mathcal{D}$ to capacity N
  Initialize action-value function Q with random weights
  $\alpha \leftarrow \alpha_0$
  for episode $= 1$, M **do**
    Initialize sequence with initial state s_1
    for $t = 1, T$ **do**
      With probability ϵ select a random action a_t, otherwise select action $a_t = \max_a Q^*(s_t, a; \theta)$
      Execute action a_t in emulator and observe reward r_t and state s_{t+1}
      Store transition (s_t, a_t, r_t, s_{t+1}) in memory $\mathcal{D}$
      Sample random minibatch of n transitions (s_j, a_j, r_j, s_{j+1}) from $\mathcal{D}$
      $y_j \leftarrow \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{for non-terminal } s_{j+1} \end{cases}$
      $g \leftarrow \frac{\partial \mathcal{L}}{\partial \theta}$
      $d \leftarrow \texttt{update\_damping}(d)$
      $g_{\text{norm}} \leftarrow \sqrt{\sum \frac{\partial L}{\partial \theta}^2}$
      Define G such that $G(v) = (\frac{1}{n}\mathbf{J}_Q v)\mathbf{J}_Q$
      Solve $\text{argmin}_x \left\| (G + d\mathbf{I})x - \frac{\partial L}{\partial \theta} \right\|$ with MinresQLP [Choi et al., 2011]
      $g_{\text{natural}} \leftarrow g_{\text{norm}} x$
      $\theta \leftarrow \theta - \alpha g_{\text{natural}}$
      $\alpha \leftarrow \Delta_\alpha \alpha$
    end for
  end for

6 Experiments

6.1 Overview

To run Q-learning models on OpenAI gym, we adapt Pascanu and Bengio's implementation [2013]. For the baseline, we use OpenAI's open-source Baselines library [Dhariwal et al., 2017], which allows reliable testing of tuned reinforcement learning architectures. As is defined in Gym, performance is measured by taking the best 100-episode reward over the course of running.

We run a grid search on the parameter spaces specified in the Hyperparameters section, measuring performance for all possible combinations. Because certain parameters like the exploration fraction are not used in our implementation of NGDQN, we grid search those parameters as well. As we wish to compare "vanilla" NGDQN to "vanilla" DQN, we do not use target nets, model saving, or any other features, such as prioritized experience replay.[2]

[2] The DQN performance shown may be worse than DQN performance one might find when comparing to other implementations. The discrepancy is that DQN is often used with target networks, whereas in this case, we

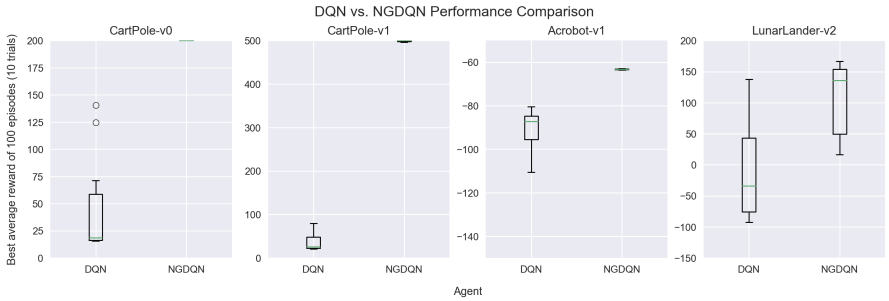

Figure 1: Average best 100-episode run over 10 trials with IQR[2]. We can see that in every environment tested, NGDQN achieves a higher max 100-episode average than our DQN baseline.

Following this grid search, we take the best result performance for each environment from both DQN and NGDQN, and run this configuration 10 times, recording a moving 100-episode average and the best 100-episode average over the course of a number of runs.

These experiments reveal that natural gradient compares favorably to standard adaptive gradient techniques. However, the increase in stability and speed comes with a trade-off: due to the additional computation, natural gradient takes longer to train when compared to adaptive methods, such as the Adam optimizer [Kingma and Ba, 2014]. Details of this can be found in Pascanu and Bengio's work [2013].

Below, we describe the environments, summarizing data taken from `https://github.com/openai/gym` and information provided on the wiki: `https://github.com/openai/gym/wiki`.

6.2 CartPole-v0

The classic control task CartPole involves balancing a pole on a controllable sliding cart on a frictionless rail for 200 timesteps. The agent "solves" the environment when the average reward over 100 episodes is equal to or greater than 195. However, for the sake of consistency, we measure performance by taking the best 100-episode average reward.

The agent is assigned a reward for each timestep where the pole angle is less than ± 12 deg, and the cart position is less than ± 2.4 units off the center. The agent is given a continuous 4-dimensional space describing the environment, and can respond by returning one of two values, pushing the cart either right or left.

6.3 CartPole-v1

CartPole-v1 is a more challenging environment which requires the agent to balance a pole on a cart for 500 timesteps rather than 200. The agent solves the environment when it gets an average reward of 450 or more over the course of 100 timesteps. However, again for the sake of consistency, we again measure performance by taking the best 100-episode average reward. This environment essentially behaves identically to CartPole-v0, except that the cart can balance for 500 timesteps instead of 200.

6.4 Acrobot-v1

In the Acrobot environment, the agent is given rewards for swinging a double-jointed pendulum up from a stationary position. The agent can actuate the second joint by returning one of three actions, corresponding to left, right, or no torque. The agent is given a six dimensional vector describing the environments angles and velocities. The episode ends when the end of the second pole is more than the length of a pole above the base. For each timestep that the agent does not reach this state, it is given a -1 reward.

---

do not use target nets for either NGDQN or DQN. The reason for this is that we want to compare NGDQN to DQN on the original DQN algorithm presented in Algorithm 1 of Mnih et al. [2013]. We plan to expand our analysis to include target networks in future work.

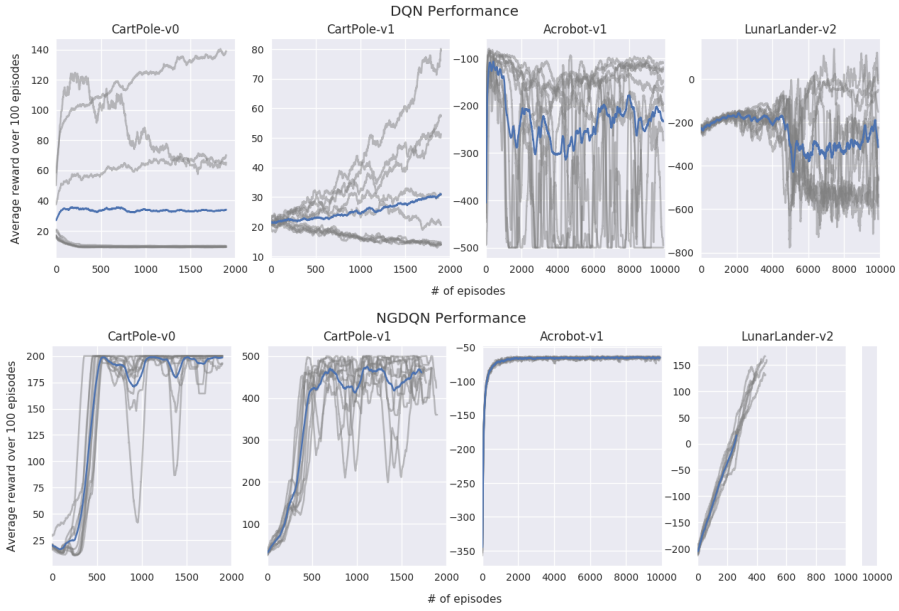

Figure 2: DQN baseline and NGDQN performance over 10 trials over time with average line[3]. We can see that when training, NGDQN appears to be significantly more stable than the DQN baseline (i.e. NGDQN teneded reliably converged to a solution while the DQN baseline did not).

6.5 LunarLander-v2

Finally, in the LunarLander environment, the agent attempts to land a lander on at a particular location on a simulated 2D world. If the lander hits the ground going too fast, the lander will explode, or if the lander runs out of fuel, the lander will plummet toward the surface. The agent is given a continuous vector describing the state, and can turn its engine on or off. The landing pad is placed in the center of the screen, and if the lander lands on the pad, it is given reward. The agent also receives a variable amount of reward when coming to rest, or contacting the ground with a leg. The agent loses a small amount of reward by firing the engine, and loses a large amount of reward if it crashes. Although this environment also defines a solve point, we use the same metric as above to measure performance.

7 Results

NGDQN and DQN were run against these four experiments, to achieve the following results summarized in Figure 1. The hyperparameters can be found in the the Hyperparameters section, and the code for this project can be found in the Code section. Each environment was run for a number of episodes (see Hyperparameters), and as per Gym standards, the best 100 episode performance was taken.

In all experiments, natural gradient converges faster and significantly more consistently than the DQN benchmark, indicating its robustness in this task compared to the standard adaptive gradient optimizer used in the Baseline library (Adam). The success across all tests indicates that natural gradient generalizes well to diverse control tasks, from simpler tasks like CartPole, to more complex tasks like LunarLander.[3]

[3]The LunarLander-v2 task for NGDQN was not completed, as the Sherlock cluster where the environments were run does not permit GPU tasks for over 48 hours. Therefore, each of the 10 trials was run for 48 hours and then stopped.

8 Conclusions

In this paper, natural gradient methods are shown to accelerate and stabilize training for common control tasks. This could indicate that Q-learning's instability may be diminished by naturally optimizing it, and also that natural gradient could be applied to other areas of reinforcement learning.

Contributions & Acknowledgements

Here, a brief contributor statement is provided, as recommended by Sculley et al. [2018].

Primary author led the research and wrote the first draft of the paper, as well as all the code for this project. Additionally, experiments were run by primary author. Secondary author verified all of the code for correctness and edited parts of the paper. Secondary author was also important in the derivation and understanding of KL divergence and the natural gradient, and in his reaching out to fellow academics.

Thanks to Jen Selby for providing valuable insight and support, and for reviewing the paper and offering her suggestions over the course of the writing process. Thanks to Leonard Pon for his instruction, advice, and generous encouragement, especially during Applied Math, where the first part of this project took place. Also, huge thanks to Kavosh Asadi for providing us with valuable feedback and direction, and for helping us navigate the research scene.

References

Shun-Ichi Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

Alex Barron, Todor Markov, and Zack Swafford. Deep q-learning with natural gradients, Dec 2016. URL `https://github.com/todor-markov/natural-q-learning/blob/master/writeup.pdf`.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *ArXiv e-prints*, abs/1606.01540, 2016. URL `http://arxiv.org/abs/1606.01540`.

Sou-Cheng T. Choi, Christopher C. Paige, and Michael A. Saunders. MINRES-QLP: A krylov subspace method for indefinite or singular symmetric systems. *SIAM Journal on Scientific Computing*, 33(4):1810–1836, 2011. doi: 10.1137/100787921. URL `http://web.stanford.edu/group/SOL/software/minresqlp/MINRESQLP-SISC-2011.pdf`.

Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, and Koray Kavukcuoglu. Natural neural networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2071–2079. Curran Associates, Inc., 2015. URL `http://papers.nips.cc/paper/5953-natural-neural-networks.pdf`.

Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. `https://github.com/openai/baselines`, 2017.

Sander Dieleman, Jan Schlüter, Colin Raffel, Eben Olson, Søren Kaae Sønderby, Daniel Nouri, Daniel Maturana, Martin Thoma, Eric Battenberg, Jack Kelly, Jeffrey De Fauw, Michael Heilman, Diogo Moitinho de Almeida, Brian McFee, Hendrik Weideman, Gábor Takács, Peter de Rivaz, Jon Crall, Gregory Sanders, Kashif Rasul, Cong Liu, Geoffrey French, and Jonas Degrave. Lasagne: First release, August 2015. URL `http://dx.doi.org/10.5281/zenodo.27878`.

Nick Foti. The natural gradient, Jan 2013. URL `https://hips.seas.harvard.edu/blog/2013/01/25/the-natural-gradient/`.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, G. Dulac-Arnold, I. Osband, J. Agapiou, J. Z. Leibo, and A. Gruslys. Deep Q-learning from Demonstrations. *ArXiv e-prints*, April 2017.

Antti Honkela, Matti Tornio, Tapani Raiko, and Juha Karhunen. Natural conjugate gradient in variational inference. *International Conference on Neural Information Processing*, 2015. URL https://www.hiit.fi/u/ahonkela/papers/Honkela07ICONIP.pdf.

Sham Kakade. A natural policy gradient. In Thomas G. Dietterich, Suzanna Becker, and Zoubin Ghahramani, editors, *Advances in Neural Information Processing Systems 14 (NIPS 2001)*, pages 1531–1538. MIT Press, 2001. URL http://books.nips.cc/papers/files/nips14/CN11.pdf.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ArXiv e-prints*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980.

S. Kullback and R. A. Leibler. On information and sufficiency. *Ann. Math. Statist.*, 22(1):79–86, 03 1951. doi: 10.1214/aoms/1177729694. URL http://dx.doi.org/10.1214/aoms/1177729694.

Long-Ji Lin. *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992.

James Martens. Deep learning via hessian-free optimization. In *ICML*, 2010.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *ArXiv e-prints*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

OpenAI. Requests for research: Initial commit, 2016. URL https://github.com/openai/requests-for-research/commit/03c3d42764dc00a95bb9fab03af08dedb4e5c547.

OpenAI. Requests for research, 2018. URL https://openai.com/requests-for-research/#natural-q-learning.

Razvan Pascanu and Yoshua Bengio. Natural gradient revisited. *ArXiv e-prints*, abs/1301.3584, 2013. URL http://arxiv.org/abs/1301.3584.

Jan Peters, Sethu Vijayakumar, and Stefan Schaal. *Natural Actor-Critic*, pages 280–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31692-3. doi: 10.1007/11564096_29. URL https://doi.org/10.1007/11564096_29.

G. A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report 166, Cambridge University Engineering Department, September 1994. URL ftp://svr-ftp.eng.cam.ac.uk/reports/rummery_tr166.ps.Z.

T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized Experience Replay. *ArXiv e-prints*, November 2015.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL http://proceedings.mlr.press/v37/schulman15.html.

D. Sculley, Jasper Snoek, Alex Wiltschko, and Ali Rahimi. Winner's curse? on pace, progress, and empirical rigor. *ICLR 2018 (under review)*, Feb 2018. URL https://openreview.net/forum?id=rJWF0Fywf.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 0262193981. URL https://pdfs.semanticscholar.org/aa32/c33e7c832e76040edc85e8922423b1a1db77.pdf.

Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3):58–68, March 1995. ISSN 0001-0782. doi: 10.1145/203330.203343. URL http://doi.acm.org/10.1145/203330.203343.

Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016. URL http://arxiv.org/abs/1605.02688.

Christopher John Cornish Hellaby Watkins. *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, May 1989. URL http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba. Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation. *ArXiv e-prints*, August 2017.

Yandex. Agentnet, 2016. URL https://github.com/yandexdataschool/AgentNet.

Appendix A: Hyperparameters

Both NGDQN and DQN had a minimum epsilon of 0.02 and had a γ of 1.0 (both default for Baselines). The NGDQN model was tested using an initial learning rate of 1.0. For NGDQN, the epsilon decay was set to 0.995, but since there wasn't an equivalent value for the Baselines library, the grid search for Baselines included an exploration fraction (defined as the fraction of entire training period over which the exploration rate is annealed) of either 0.01, 0.1, or 0.5. Likewise, to give baselines the best chance of beating NGDQN, we also searched a wide range of learning rates, given below.

Environment	# of Episodes Ran For	Layer Configuration
CartPole-v0	2000	[64]
CartPole-v1	2000	[64]
Acrobot-v1	10,000	[64, 64]
LunarLander-v2	10,000	[256, 128]

Table 1: Shared configuration

The batch job running time is given below (hours:minutes:seconds) for Sherlock. NGDQN LunarLander-v2 was run on the gpu partition which supplied either an Nvidia GTX Titan Black or an Nvidia Tesla GPU. All other environments were run on the `normal` partition. Additionall details about natural gradient computation time can be found in Pascanu and Bengio [2013].

Environment	NGDQN Batch Time	DQN Batch Time
CartPole-v0	4:00:00	1:00:00
CartPole-v1	9:00:00	1:00:00
Acrobot-v1	48:00:00	8:00:00
LunarLander-v2	48:00:00[4]	12:00:00

Table 2: Running time

Best grid searched configuration used in both the DQN and NGDQN experiments:

[4]Jobs not completed; see Figure 2 for details

Environment	Natural Gradient DQN (NGDQN)					Baseline DQN				
	Learning Rate	Adapt Damping	Batch Size	Memory Length	Activ-ation	Learning Rate	Exploration fraction	Batch Size	Memory Length	Activ-ation
CartPole-v0	0.01	No	128	50,000	Tanh	1e-07	0.01	128	2500	Tanh
CartPole-v1	0.01	Yes	128	50,000	Tanh	1e-08	0.1	32	50,000	Tanh
Acrobot-v1	1.0	No	128	50,000	Tanh	1e-05	0.01	128	50,000	ReLU
LunarLander-v2	0.01	No	128	50,000	ReLU	1e-05	0.01	128	2500	Tanh

NGDQN hyperparameter search space (total # of combos: 48):

Hyperparameter	Search Space
Learning Rate	[0.01, 1.0]
Adapt Damping	[Yes, No]
Batch Size	[32, 128]
Memory Length	[500, 2500, 50000]
Activation	[Tanh, ReLU]

DQN hyperparameter search space (total # of combos: 360):

Hyperparameter	Search Space
Learning Rate	[1e-08, 1e-07, 1e-06, 5e-06, 1e-05, 5e-05, 0.0001, 0.0005, 0.005, 0.05]
Exploration Fraction	[0.01, 0.1, 0.5]
Batch Size	[32, 128]
Memory Length	[500, 2500, 50000]
Activation	[Tanh, ReLU]

Appendix B: Code

The code for this project can be found at `https://github.com/hyperdo/natural-gradient-deep-q-learning`. It uses a fork of OpenAI Baselines to allow for different activation functions: `https://github.com/hyperdo/baselines`.