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

We present a novel algorithm to train a deep Q-learning agent using natural-gradient techniques. We compare the original deep Q-network (DQN) algorithm to its natural-gradient counterpart, which we refer to as NGDQN, on a collection of classic control domains. Without employing target networks, NGDQN significantly outperforms DQN without target networks, and performs no worse than DQN with target networks, suggesting that NGDQN stabilizes training and can help reduce the need for additional hyperparameter tuning. We also find that NGDQN is less sensitive to hyperparameter optimization relative to DQN. Together these results suggest that natural-gradient techniques can improve value-function optimization in deep reinforcement learning.

1 Introduction

A core piece of various reinforcement-learning algorithms is the estimation of a value function– the expected sum of future discounted rewards under a desired policy [Sutton and Barto, 1998]. Q-learning is a model-free reinforcement learning algorithm that estimates the value function under the optimal policy by minimizing the temporal-difference error between the agent's value function estimates [Watkins, 1989]. This basic algorithm, when combined with deep neural networks [LeCun et al., 2015], has proven to be a major success in AI, most notably by exhibiting human-level performance in a suite of challenging Atari games [Mnih et al., 2013].

Because Q-learning, as the control extension of TD algorithms [Sutton, 1988], is not truly a stochastic gradient descent algorithm [Maei, 2011], convergence of the algorithm with non-linear function approximators is poorly understood. In fact, it has been shown that TD algorithms can sometimes be divergent [Tsitsiklis and Van Roy, 1997]. Moreover, in practice it is sometimes difficult to train a deep neural network with Q-learning. In the original DQN work, for example, the authors proposed three key additions to stabilize training, namely experience replay [Lin, 1992], reward clipping, and the use of target networks. [Mnih et al., 2013]

In this work, we aim to address some of the practical issues pertaining to DQN training as well as improve upon it by using natural-gradient techniques. Natural gradient was originally proposed by Amari as a method to accelerate gradient descent [1998]. Rather than exclusively using the loss gradient or using local curvature from the Hessian matrix, natural gradient uses "information" found in the parameter space of the model to train efficiently.

Natural gradient has been successfully applied to several deep learning domains [Desjardins et al., 2015, Schulman et al., 2015, Wu et al., 2017] and has been used to accelerate the training of reinforcement learning systems [Kakade, 2001, Peters and Schaal, 2008, Dabney and Thomas, 2014]. To motivate our approach, we hoped that using natural gradient would accelerate the training of DQN, 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,

Preprint. Work in progress.

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 its training appeared much more stable. Further, NGDQN performed no worse than DQN with target networks trained over the same number of episodes.

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

2.1 Reinforcement learning problem

In the reinforcement learning problem, typically modeled as an MDP [Puterman, 2014], an agent interacts with an environment to maximize cumulative reward. The agent observes a state s, performs an action a, and receives a new state s' and reward r. Usually a discount factor γ is also defined, which specifies the relative importance of immediate reward as opposed to those recieved in the future. More specifically the objective is to maximize:

$$E_\pi[R_t] = E\big[\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} | \pi\big] \,, \tag{1}$$

by attempting to learn a good policy π.

2.2 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 $\pi(s) = \text{argmax}_a Q^*(s, a)$ [Sutton and Barto, 1998, Hester et al., 2017]. 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 with a linear activation on the final layer. The loss of this network is defined as follows, given the environment ε:

$$\mathcal{L} = \mathbb{E}\left[(y - Q(s, a_i; \theta))^2\right] \tag{3}$$

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

$$y = \mathbb{E}_{s' \sim \varepsilon}\left[r + \gamma \max_{a'} Q(s', a'; \theta) \big| s, a_i\right] \tag{4}$$

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 in which an agent successfully played Atari games from pixels, an adaptive gradient method was 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, DQN uses an ϵ-greedy policy: the agent acts nearly randomly in order to explore potentially successful strategies, and as the agent learns, it acts randomly less often (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 annealed over the course of training.

We combine these two approaches, using natural gradient to optimize the 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 updates parameters θ of a model mapping from x to y as $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(x, y; \theta)$ given a learning rate of α.

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]:

$$KL(\mu_1|\mu_2) = \int_{-\infty}^{\infty} \mu_1(s) \log \frac{\mu_1(s)}{\mu_2(s)} \, \mathrm{d}s \tag{5}$$

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

$$KL_{\text{sym}}(\mu_1|\mu_2) := \frac{1}{2} \left(KL(\mu_1|\mu_2) + KL(\mu_2|\mu_1) \right) \tag{6}$$

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]:

$$\mathbf{F}_\theta = \mathbb{E}_{s,q}[(\nabla \log p_\theta(q|s))^T (\nabla \log p_\theta(q|s))] \tag{7}$$

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]:

$$\nabla \mathcal{L}_N = \nabla \mathcal{L} {\mathbf{F}_\theta}^{-1} \tag{8}$$

Using this definition and solving the Lagrange multiplier for minimizing the loss of parameters updated by $\Delta\theta$ under the constraint of a constant symmetric KL divergence, one can derive the

approximation for constant symmetric KL divergence, using the information matrix. Taking the second-order Taylor expansion, Pascanu and Bengio [2013] derive:

$$KL_{\text{sym}}(p_\theta | p_{\theta+\Delta\theta}) \approx \frac{1}{2}\Delta\theta^T \mathbf{F}_\theta \Delta\theta \tag{9}$$

As the output probability distribution is dependent on the final layer activation, Pascanu and Bengio [2013] give the following representation 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:

$$p_\theta(q|s) = \mathcal{N}(q|Q(s,\theta), \beta^2) \tag{10}$$

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 with respect to the parameters as follows:

$$\mathbf{F}_{\text{linear}} = \beta^2 \mathbb{E}_{s\sim d^\pi(s)} \left[\mathbf{J}_Q^T \mathbf{J}_Q\right] \tag{11}$$

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 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 attempt of natural Q-learning was created by [Barron et al., 2016]. In this work, the authors re-implemented PRONG and verified its efficacy at MNIST. However, when the authors tried 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 adapted from Pascanu and Bengio originally fit an X to a mapping y and directly back-propagated a loss, we modify the training procedure to use a target value change similar to that described in equation 3. 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 11, approximated at every batch.

We calculate the desired 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 with respect to the weights: $Gx = \frac{\partial L}{\partial \theta}$ (see Algorithm 1). The MinRes-QLP Algorithm solves this linear equation by extending MinRes, an existing Krylov subspace descent algorithm to solve linear equations, to ill-conditioned systems such as a singular FIM using the QLP decomposition of the tridiagonal matrix from the Lanczos process [Choi et al., 2011]. This method finds minimum length solutions robust to different conditions. We also test Linear Conjugate Gradient, an algorithm that solves the linear equation by decomposing x into vectors conjugate with respect to G and iteratively calculating its components. Linear Conjugate Gradient is used for solving linear equations quickly and efficiently, with $\mathcal{O}(m\sqrt{k})$ where m is the number of nonzero entries of G and k is its condition number [Shewchuk, 1994].

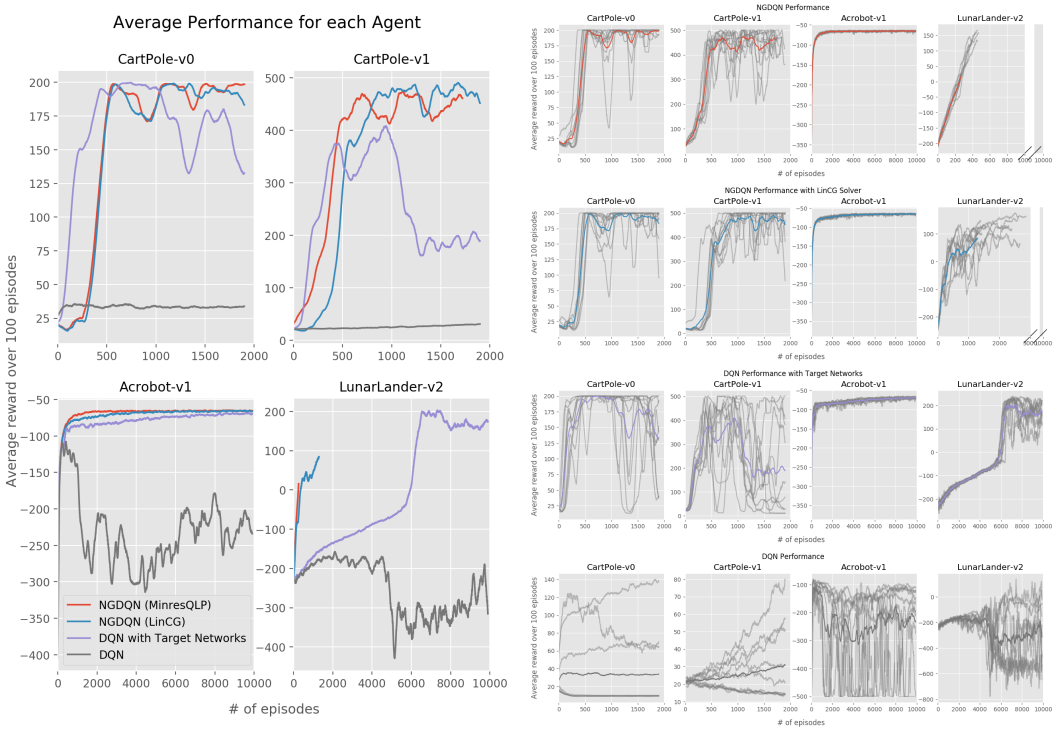
For both algorithms, a damping factor d is applied to ensure computability: $G := G + d\mathbf{I}$. Another efficiency of using linear solvers is that we are able to represent the FIM as an operator on x without needing to explicitly compute the matrix. We take advantage of this by using Theano's left and right operator representations (Lop and Rop) of the Jacobian, as well as compare the linear solvers to Theano's explicit matrix inversion. This inversion utilizes the Gauss–Jordan elimination method to invert the Fisher information matrix with asymptotic time complexity $\mathcal{O}(n^3)$ [Theano Development Team, 2016].

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, $\Delta\alpha$ 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 $\mathcal{D}$
      $y_j \leftarrow \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{for non-terminal } s_{j+1} \end{cases}$
      $g \leftarrow \frac{\partial \mathcal{L}}{\partial \theta}$
      $d \leftarrow$ `update_damping`(d)
      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 linear solver (e.g. MinresQLP [Choi et al., 2011])
      $\theta \leftarrow \theta - \alpha x$
      $\alpha \leftarrow \Delta_\alpha \alpha$
    end for
  end for



(a) (b)

Figure 1: NGDQN and DQN performance over 10 trials over time with average line[1]. We can see that when training, NGDQN appears to be significantly more stable than the DQN baseline (i.e. NGDQN tended to reliably converge to a solution while the DQN baseline without target nests did not).

6 Experiments

6.1 Control Tasks

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 test a version where target networks, model saving, or any other features, such as prioritized experience replay are not used. To further test the capabilities of NGDQN, we also compare NGDQN (which in these experiments are always run without target networkss) to DQN with target networks in order to show that the algorithm is competitive with other stabilization techniques.

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 mean best 100-episode average across each run. These experiments reveal that NGDQN without target networks compare favorably to standard adaptive gradient techniques, even outperforming DQN with target networks. 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].

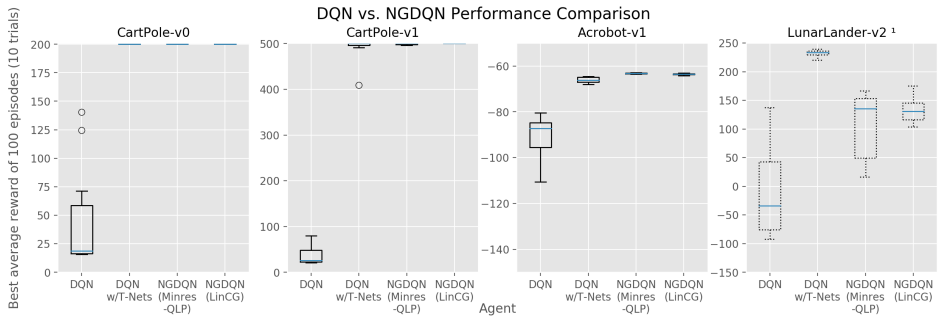


Figure 2: Average best 100-episode run over 10 trials with IQR. We can see that in every environment fully run, NGDQN achieves a higher max 100-episode average than our DQN baselines.[1]

We test NGDQN in this manner on four common control environments from `https://github.com/openai/gym`: CartPole-v0, CartPole-v1, Acrobot-v1, and LunarLander-v2 (see Appendix B).

6.2 Inversion Methods

During training, updates to the weights are calculated by solving a linear system (see Algorithm 1), equivalent to matrix-vector-product of the inverse damped FIM with the gradient of the loss. We test the different methods to solve for these updates by computing the parameter updates from both MinRes-QLP and Linear Conjugate Gradient and comparing them to the updates given by an explicitly computed true FIM inversion. After separately solving for these individual updates, we calculate a variety of metrics to record how the natural gradient differs between inversion methods. This process is measured over 100 episodes of training on CartPole-v0, using the Linear CG's parameter updates.

For this NGDQN algorithm to satisfy the theoretical properties of natural gradient, an accurate inversion method is needed, and in order to create an effective yet efficient algorithm, it is necessary to balance accuracy and computational cost. Because natural gradient alters the step size of the gradient descent vector according to second order information and the angle of that vector through KL divergence, we record the norm of the calculated natural gradient and the angle between the update steps between true values solvers and estimators.[2] We also record the computation time of each method. Finally, to ensure that damping is not significantly skewing the NG calculation, we compute the maximal eigenvalue of the Fisher information matrix by optimizing $\max_{\hat{x}} \hat{x}^T G \hat{x}$ (see Appendix C), which gives us an indication of the scaling done by the FIM. By comparing this value to the damping factor, we see that damping is relatively small and not significantly skewing the calculation of the natural gradient.

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 Appendix A, and the code for this project can be found in the Appendix D. Each environment was run for a number of episodes (see Appendix A), and as per Gym standards the best 100 episode performance was taken.

In all experiments, natural gradient converges faster and achieves higher performance 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). NGDQN also arrives at better solutions more reliably across the searched hyperparameters, exhibiting its versatility to different configurations when compared to the harsh tuning of DQN (see Appendix A). The success across all

[1]The LunarLander-v2 task for NGDQN was not completed, as the Stanford 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.

[2]Calculated as $\arccos(\hat{a} \cdot \hat{b})$, where $\hat{a}$ is the flattened normalized updates computed by the true inverse and $\hat{b}$ is the flattened normalized updates given by the linear solvers.

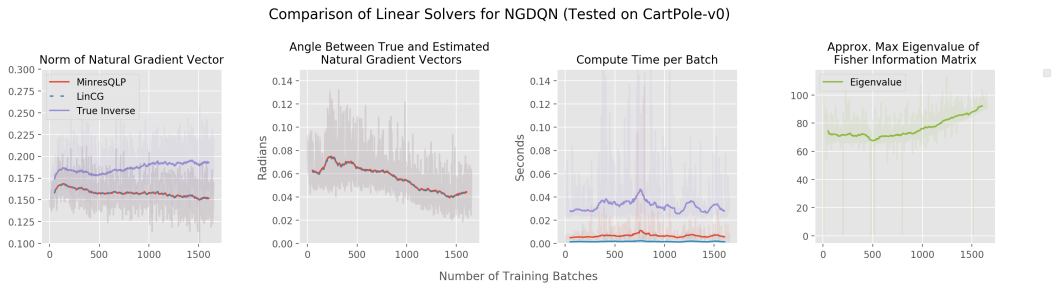


Figure 3: Summary of comparisons between different inversion methods. Moving averages of 100-batches are provided for readability, marked with solid lines.[3]

tests indicates that natural gradient generalizes well to diverse control tasks, from simpler tasks like CartPole to more complex tasks like LunarLander.

Comparison of the different inversion methods reveals that they calculate similar parameter updates. We find that MinRes-QLP and Linear CG arrive at updates with slightly smaller magnitudes and extremely similar directions as computing the true matrix inversion of the FIM, indicating that even with these approximations, the algorithm is consistent with the theory behind natural gradient. It is also shown that the damping factor is most often between 5% to 10% of the maximal eigenvalue, indicating that the FIM is not over-damped. Finally, the compute times for the estimated FIM inversions are shown to be significantly less than that of the true FIM inversions, showing that using these methods helps accelerate training.

8 Discussion

In this paper, natural-gradient methods are shown to accelerate and stabilize training for common control tasks, even without target networks. 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 in order to address important problems such as sample efficiency.

Although we have not yet empirically investigated the precise cause of this increase in stability, we offer a possible explanation below. One potential cause is that, although the replay buffer partially decorrelates the training set and time, the replay buffer will, over the course of training, become more filled with transitions from later on in each environment's episode.

When playing, our agent's buffer will, during the beginning of training, be primarily filled with transitions from when the agent was acting poorly (e.g. in LunarLander-v2, the replay buffer is filled with transitions where the craft eventually plummets to a fiery death). However, much later in training when the agent has learned a policy to achieve higher rewards, the overall composition of this training buffer will be shifted to become more relevant to what the agent has to learn later in training. As the first few steps of gradient descent have a disproportionately large impact on the trained model, training with SGD could potentially be destabilized later on by these largely random transitions Pascanu and Bengio [2013].

In this scenario, target networks could help stabilize training. Natural gradient, by comparison, is very robust to reordering of the training set [Pascanu and Bengio, 2013]. This means that NGDQN could potentially use experience acquired later in training more effectively, as the overall policy of the agent would not be as skewed to experience gained early during training. This is, of course, only one possible explanation, and we hope that researchers will further investigate this phenomena in later work.

Contributions & Acknowledgements

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

[3]Tested on separate computer, hence slightly different compute time from agent training.

Primary author led and directed the research, wrote the paper, edited the paper, wrote/adapted all the learning code for this project, and ran the RL experiments. Secondary author wrote and performed the inversion experiments, wrote parts of the paper, debugged and verified all of the code for correctness, and was important in editing the paper.

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. Thanks to Joshua Achiam for the valuable discussions, and for imparting his support, advice, and know-how. Also, huge thanks to Kavosh Asadi for providing us with valuable guidance, feedback, edits, 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.

William Dabney and Philip S Thomas. Natural temporal difference learning. *AAAI Conference on Artificial Intelligence*, 2014. URL https://www.aaai.org/ocs/index.php/AAAI/AAAI14/paper/view/8568.

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.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

Hamid Reza Maei. Gradient temporal-difference learning algorithms. *ACM Digital Library*, 2011. URL https://dl.acm.org/citation.cfm?id=2518887.

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 and Stefan Schaal. Natural actor-critic. *Neurocomputing*, 71(7-9):1180–1190, 2008.

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.

Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

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.

J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, 1994.

Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3 (1):9–44, 1988.

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.

John N Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997.

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 (see Table 4). 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. Additional 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

Hyperparameter grid-search space:

Hyperparameter	Search Space
Learning Rate	[0.01, 0.1, 1.0]
Adapt Damping	[Yes, No]
Batch Size	[32, 128]
Memory Length	[2500, 50000]
Activation	[Tanh, ReLU]

Table 3: NGDQN hyperparameter search space

[4]Jobs not completed; see Figure 1 for details

Hyperparameter	Search Space
Learning Rate (no TNs)	[1e-08, 1e-07, 1e-06, 5e-06, 1e-05, 5e-05, 0.0001, 0.0005, 0.005, 0.05]
Learning Rate (with TNs)[5]	[1e-08, 1e-07, 1e-06, 1e-05, 1e-04, 1e-03]
Exploration Fraction	[0.01, 0.1, 0.5]
Batch Size	[32, 128]
Memory Length	[500, 2500, 50,000]
Activation	[Tanh, ReLU]
Target Network Update Freq	[N/A, 500, 1000, 10,000]

Table 4: Baseline DQN hyperparameter search space

Best grid-searched configurations, used for experiments:

Environment	Learning Rate	Exploration fraction	Batch Size	Memory Length	Activ-ation
CartPole-v0	1e-07	0.01	128	2500	Tanh
CartPole-v1	1e-08	0.1	32	50,000	Tanh
Acrobot-v1	1e-05	0.01	128	50,000	ReLU
LunarLander-v2	1e-05	0.01	128	2500	Tanh

Table 5: Baseline DQN hyperparameter configuration

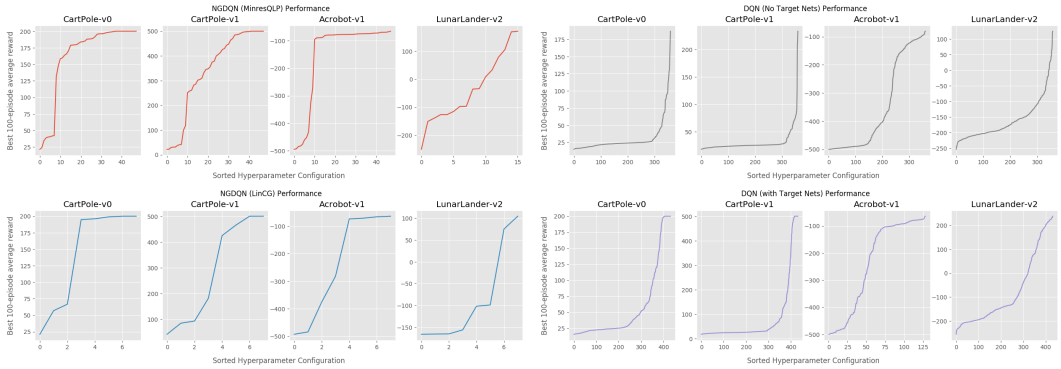
Environment	Learning Rate	Exploration Fraction	Batch Size	Memory Length	Target Net Update Freq	Activ-ation
CartPole-v0	0.001	0.01	128	50,000	500	Tanh
CartPole-v1	0.001	0.01	32	50,000	500	Tanh
Acrobot-v1	0.001	0.01	32	50,000	500	Tanh
LunarLander-v2	0.0001	0.1	128	50,000	10,000	ReLU

Table 6: Baseline DQN with target nets hyperparameter configuration

Environment	Learning Rate	Adapt Damping	Batch Size	Memory Length	Activ-ation
CartPole-v0	0.01	No	128	50,000	Tanh
CartPole-v1	0.01	Yes	128	50,000	Tanh
Acrobot-v1	1.0	No	128	50,000	Tanh
LunarLander-v2	0.01	No	128	50,000	ReLU

Table 7: NGDQN (MinresQLP) hyperparameter configuration

[5]Due to training idiosyncrasies, the learning rate search space was different and some configurations for Acrobot-v1 were not run, although we believe given the results, this minor difference is insignificant



(a) (b)

Figure 4: Grid searched performances over tested hyperparameter configurations for NGDQN (left) and DQN (right), ordered by increasing performance, hinting at robustness to changing hyperparameters

Environment	Learning Rate	Adapt Damping	Batch Size	Memory Length	Activation
CartPole-v0	0.01	N/A	128	50,000	Tanh
CartPole-v1	0.01	N/A	128	50,000	Tanh
Acrobot-v1	0.1	N/A	128	50,000	Tanh
LunarLander-v2	0.01	N/A	128	50,000	Tanh

Table 8: NGDQN (LinCG) hyperparameter configuration [6]

Appendix B: Environments

Data is summarized from `https://github.com/openai/gym` and information provided on the wiki: `https://github.com/openai/gym/wiki`.

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.

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] Damping adapt, learning rate 1.0, memory length 500, batch size 32 not tested for LinCG due to poor results from initial tests to reduce computation burden

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.

LunarLander-v2

Finally, in the LunarLander environment, the agent attempts to land a lander on 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.

Appendix C: Computation of the Maximal Eigenvalue

To ensure our inversion is not overdamped, we compare the maximal eigenvalue the Fisher information matrix to its damping factor. To calculate this eigenvalue, we optimize $\max_x \hat{x}^T G\hat{x}$, as our FIM is implemented as a matrix-vector product only. Here we give pseudocode to calculate the eigenvector and outline a proof to show that this method has a global at minimum the maximal eigenvalue.

Algorithm 2 Algorithm to find the approximate maximum eigenvalue

Require: Matrix vector product Gx given x
Require: Starting vector v, initialized randomly
Require: Early stopping condition (c: 0.001)
Require: Training steps (steps: 10000)
Require: Learning rate (α: 0.0005)
  for $i = 1$, steps **do**
    $\Delta v \leftarrow \nabla_v \left(\frac{v}{\|v\|} \cdot G \left(\frac{v}{\|v\|}^T \right) \right)$
    $v \leftarrow v + \alpha \Delta v$
    if $\|\Delta v\|_2 > c$ **then**
      break
    end if
  end for
  return $\frac{v}{\|v\|} \cdot G \left(\frac{v}{\|v\|}^T \right)$

To prove this is maximized at the maximal eigenvalue of G, we show that $\hat{x}^T G\hat{x}$ is equivalent to the dot product $\hat{x} \cdot G\hat{x}$, which is also expressed as $\|\hat{x}\| \, \|G\hat{x}\| \cos\theta$ (where θ is the angle between $\hat{x}$ and $G\hat{x}$).

Clearly, $\|\hat{x}\| = 1$. Then, $\|Gx\|$ is maximized at the maximal eigenvalue of G. This is because any $\hat{x}$ can be decomposed into $\sum_{i=1}^{N_\lambda} c_i \hat{v}_i$ where $\hat{v}_i$'s are the unit eigenvectors of G. When $\hat{x}$ is operated by G,

$$G\hat{x} = G \sum_{i=1}^{N_\lambda} c_i \hat{v}_i = \sum_{i=1}^{N_\lambda} c_i G\hat{v}_i = \sum_{i=1}^{N_\lambda} \lambda_i c_i \hat{v}_i \tag{12}$$

Thus, $\|G\hat{x}\|$ is maximized at $\|G\hat{x}\| = \lambda_{\text{max}}$ when $\hat{x} = \hat{v}_{\text{max}}$. Furthermore, $\cos\theta$ is maximized at $\theta = 0$, which is also true when $\hat{x} = \hat{v}_{\text{max}}$ since $\hat{v}_{\text{max}}$ and $G\hat{v}_{\text{max}}$ are in the same direction by the

definition of an eigenvector. Since each of the factors are maximized at this point, we thus show that the expression $\hat{x}^T G \hat{x}$ is maximized when $\hat{x} = \hat{v}_{\text{max}}$. It's maximal value is then

$$\hat{v}_{\text{max}}^T G \hat{v}_{\text{max}} = \hat{v}_{\text{max}}^T (\lambda_{\text{max}} \hat{v}_{\text{max}}) = \lambda_{\text{max}} (\hat{v}_{\text{max}}^T \hat{v}_{\text{max}}) = \lambda_{\text{max}} \tag{13}$$

Appendix D: 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`.