Deep Natural Gradient Q-learning

Ethan Knight
ethan.h.knight@gmail.com
The Nueva School

Osher Lerner
osherler@gmail.com
The Nueva School

January 29, 2018

Abstract

This paper present findings for training a Q-learning reinforcement learning model using natural gradient techniques. When comparing the original "vanilla" DQN algorithm to its natural gradient counterpart (NGDQN), experiments on classic controls environments show that NGDQN compared favorably to DQN in standard control task benchmarks, indicating that natural gradient could be extended to other reinforcement learning applications to accelerate 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.

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 Natural gradient

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 travel in the opposite direction of that gradient (Goodfellow, Bengio, & Courville, 2016). Mathematically, gradient descent proposes the point with a learning rate of ϵ:

$$x - \epsilon \nabla_x f(x) \tag{1}$$

However, this approach 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).

$$KL(P|Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x, \tag{2}$$

However, since $KL(P|Q) \neq KL(Q|P)$, we define symmetric KL divergence as follows (Foti, 2013):

$$KL(P|Q) := \frac{1}{2} \left(KL(P|Q) + KL(Q|P) \right) \tag{3}$$

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, we derive a Riemannian metric as the Hessian of the divergence of symmetric KL divergence. We give Pascanu and Bengio's definition (2013): given some probability density function p and parameters θ:

$$\mathbf{F}_\theta = \mathbb{E}[(\nabla \log p_\theta(z))^T (\nabla \log p_\theta(z))] \tag{4}$$

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):

$$\nabla \mathcal{L}_N = \nabla \mathcal{L} {\mathbf{F}_\theta}^{-1} \tag{5}$$

Taking the second-order Taylor expansion (Pascanu & Bengio, 2013) gets:

$$KL(p_\theta | p_{\theta + \Delta\theta}) \approx \frac{1}{2} \Delta\theta^T \mathbf{F}_\theta \Delta\theta \tag{6}$$

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 give the following derivations for each output layer type (2013):

$$\mathbf{F}_{linear} = \beta^2 \mathbb{E}_{X \sim \tilde{q}} \left[\mathbf{J}_y^T \mathbf{J}_y \right] \tag{7}$$

$$\mathbf{F}_{sigmoid} = \left[\mathbf{J}_y^T \mathrm{diag} \left(\frac{1}{y(1-y)} \right) J_y \right] \tag{8}$$

We use the first derivation (linear) in our formulation for natural gradient Q-learning.

2.2 Q-learning

Q-learning is a model-free reinforcement learning technique which works by gradually learning an action-value function, called Q (Mnih et al., 2013). For a state s, action a, and learning rate α, the function Q estimates the future reward, which is estimated using its current model's best predicted outcome. Its update rule is given as follows:

$$Q_{k+1}(s, a) \leftarrow (1 - \alpha) Q_k(s, a) + \alpha \mathrm{target} \tag{9}$$

If the agent has not yet reached the end of the task, given discount factor γ we define the target as

$$\mathrm{target} = \mathrm{reward} + \gamma \max_{a'} Q_k(s', a') \tag{10}$$

Essentially, our policy picks the best action that maximizes future reward. At the final timestep, since there is no future reward, target = 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 in order to adjust the parameters according to their effect on the loss.

For deep Q-learning, we define the neural network loss as follows :

$$L_i(\theta_i) = \mathbb{E}_{x,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right] \tag{11}$$

where

$$y_i = \mathbb{E}_{s'\sim\varepsilon}\left[r + \gamma \text{max} Q^*(s',a';\theta_{i-1})\middle|s,a\right] \tag{12}$$

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).

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 "explore" stage). Mathematically, the probability of choosing a random action ϵ is gradually linearly annealed over the course of training.

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, 1993). This serves to smooth the learning process.

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

3 Related work

We borrow heavily from Pascanu and Bengio's approach, using their natural gradient for deep neural networks formalization and implementation in our method (2013).

Next, we look at Desjardins, Simonyan, Pascanu, and Kavukcuoglu's work on a different method of natural gradient descent (2015). In this paper, we propose an algorithm called "Projected Natural Gradient Descent" (PRONG), 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).

In Kakade's (2001) and Peters, Vijayakumar, and Schaal's work (2005), additional methods of applying natural gradient to reinforcement learning are explored, with applications to policy gradient, and actor-critic, a continuous variant of the Q-learning algorithm. 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.

Honkela, Tornio, Raiko, and Karhunen's work provides insight into the mathematics of optimization using natural conjugate gradient techniques (2015). These methods allow for more efficient optimization in high dimensions and nonlinear contexts.

Finally, to our knowledge, the only one other published or publicly available implementation of natural Q-learning was created by Barron, Markov, and Swafford (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.

4 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 (10). 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 need to use the parameterization of the Fisher information matrix for linear activations, which determines the natural gradient. For this, we refer to Pascanu and Bengio's formula listed in eq. 26 (2013). Given a neural network it is assumed that

$$p_\theta(t|x) = \prod_{i=1}^{o} \mathcal{N}(t_i|y(x,\theta)_i, \beta^2) \tag{13}$$

$$\mathbf{F} = \beta^2 \mathbb{E}_{x\sim\tilde{q}}\left[\mathbf{J}_\mathbf{y}^T \mathbf{J}_\mathbf{y}\right] \tag{14}$$

The MinRes-QLP Krylov Subspace Descent Algorithm (Choi, Paige, & Saunders, 2011) solves the change in parameters, as in (Pascanu & Bengio, 2013). Our implementation runs on the OpenAI Gym platform which provides several classic control environments, such as the ones shown here (Brockman et al., 2016). The current algorithm takes a continuous space and maps it to a discrete set of actions.

Below, 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.

Algorithm 1: Deep Natural Q-Learning with Experience Replay

Parameters: Learning rate ($\alpha_0 : 1.0$), learning rate decay ($\Delta_\alpha : 0.9998$), function "update_damping"
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
Initialize α to α_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 $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } s_{j+1} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta) & \text{for non-terminal } s_{j+1} \end{cases}$
    Set $g = \frac{\partial \mathcal{L}}{\partial \theta}$
    Set $d =$ update_damping(d)
    Set $g_{\text{norm}} = \sqrt{\sum \frac{\partial L}{\partial \theta}^2}$
    Define G such that $G(v) = (\frac{1}{g_i}\mathbf{J}_Q v)\mathbf{J}_Q$
    Solve $\text{argmin}_x \left\|(G + d\mathbf{I})x - \frac{\partial L}{\partial \theta}\right\|$ with Minres QLP (Choi et al., 2011)
    Set $g_{\text{natural}} = g_{\text{norm}} x$
    Set $\theta = \theta - \alpha g_{\text{natural}}$
    Set $\alpha = \Delta_\alpha \alpha$
  end
end

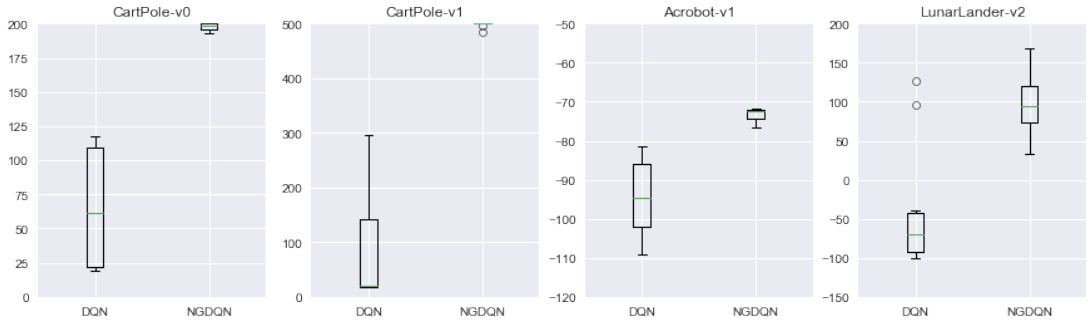

Figure 1: Average best 100-episode run over 10 trials with IQR[1]

5 Experiments

5.1 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 (OpenAI, 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 Appendix, 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 (details in Appendix). 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.[1]

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 (detailed in Appendix).

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 & 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`.

5.2 CartPole-v0

The classic control task CartPole involves balancing a pole on a controllable sliding cart on a friction-less rail for 200 timesteps. The agent "solves" the environment when the

[1]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 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.

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.

5.3 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.

5.4 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.

5.5 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.

6 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.

The code for this project can be found at `https://github.com/hyperdo/natural-gradient-deep-q-learning`. It currently includes the baselines, and we plan to post the NGDQN code as well very soon. It uses a fork of OpenAI Baselines to allow for different activation functions: `https://github.com/hyperdo/baselines`. Each environment was run for a number of episodes (see Appendix), and as per Gym standards, the best 100 episode performance was taken.

7 Conclusions

In most experiments, natural gradient converges faster and significantly more consistently than the DQN benchmark, indicating its robustness in this task compared to the standard

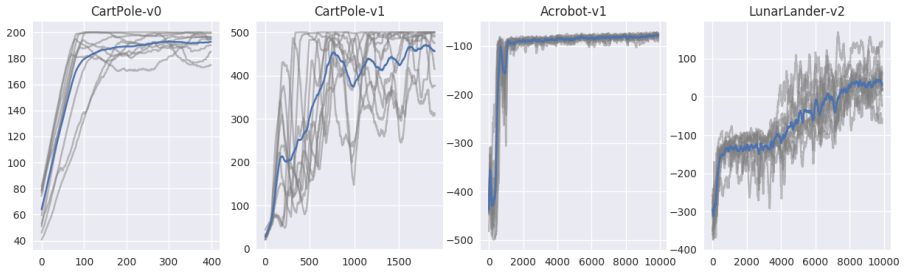

Figure 2: NGDQN performance over 10 trials with average line

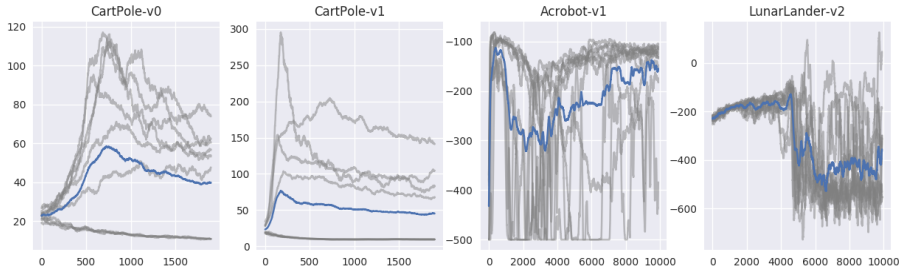

Figure 3: DQN performance over 10 trials with average line

adaptive gradient optimizer used in the Baseline library (Adam). The success across all tests indicates that natural gradient generalizes well to diverse control tasks. In LunarLander in particular, the NGDQN was able to find solutions where the DQN barely showed improvement.

In this paper, natural gradient methods are shown to accelerate 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.

8 Acknowledgements

We are deeply grateful to to Jen Selby and Leonard Pon for their instruction, advice, and generous encouragement. Thanks to Osher for his constant help and for verifying the correctness of the code and paper.

References

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

Barron, A., Markov, T., & Swafford, Z. (2016, Dec). *Deep q-learning with natural gradients.* Retrieved from `https://github.com/todor-markov/natural-q-learning/blob/master/writeup.pdf`

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., & Zaremba, W. (2016). Openai gym. *CoRR*, *abs/1606.01540*. Retrieved from `http://arxiv.org/abs/1606.01540`

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

Desjardins, G., Simonyan, K., Pascanu, R., & Kavukcuoglu, K. (2015). Natural neural networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems 28* (pp. 2071–2079). Curran Associates, Inc. Retrieved from http://papers.nips.cc/paper/5953-natural-neural-networks.pdf

Dieleman, S., Schlüter, J., Raffel, C., Olson, E., Sønderby, S. K., Nouri, D., ... Degrave, J. (2015, August). *Lasagne: First release.* Retrieved from http://dx.doi.org/10.5281/zenodo.27878 doi: 10.5281/zenodo.27878

Foti, N. (2013, Jan). *The natural gradient.* Retrieved from https://hips.seas.harvard.edu/blog/2013/01/25/the-natural-gradient/

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning.* MIT Press. (http://www.deeplearningbook.org)

Honkela, A., Tornio, M., Raiko, T., & Karhunen, J. (2015). Natural conjugate gradient in variational inference. Retrieved from https://www.hiit.fi/u/ahonkela/papers/Honkela07ICONIP.pdf

Kakade, S. (2001). A natural policy gradient. In T. G. Dietterich, S. Becker, & Z. Ghahramani (Eds.), *Advances in neural information processing systems 14 (nips 2001)* (p. 1531-1538). MIT Press. Retrieved from http://books.nips.cc/papers/files/nips14/CN11.pdf

Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, *abs/1412.6980*. Retrieved from http://arxiv.org/abs/1412.6980

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

Lin, L.-J. (1993, 01). Reinforcement learning for robots using neural networks.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, *abs/1312.5602*. Retrieved from http://arxiv.org/abs/1312.5602

OpenAI. (2016). *Requests for research: Initial commit.* Retrieved from https://github.com/openai/requests-for-research/commit/03c3d42764dc00a95bb9fab03af08dedb4e5c547

OpenAI. (2017, May). *Openai baselines.* Retrieved from https://github.com/openai/baselines

OpenAI. (2018). *Requests for research.* Retrieved from https://openai.com/requests-for-research/#natural-q-learning

Pascanu, R., & Bengio, Y. (2013). Natural gradient revisited. *CoRR*, *abs/1301.3584*. Retrieved from http://arxiv.org/abs/1301.3584

Peters, J., Vijayakumar, S., & Schaal, S. (2005). Natural actor-critic. In J. Gama (Ed.), *Machine learning: Ecml 2005: 16th european conference on machine learning, porto, portugal, october 3-7, 2005. proceedings* (pp. 280–291). Berlin, Heidelberg: Springer Berlin Heidelberg. Retrieved from https://doi.org/10.1007/11564096_29 doi: 10.1007/11564096_29

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

Retrieved from http://arxiv.org/abs/1605.02688

Yandex. (2016). *Agentnet.* Retrieved from https://github.com/yandexdataschool/AgentNet

9 Appendix

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]

NGDQN hyperparameter search space:

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

DQN hyperparameter search space:

Hyperparameter	Search Space
Learning rate	[1e-6, 5e-6, 1e-5, 5e-5, 1e-4, 5e-4, 5e-3, 5e-2]
Exploration Fraction	[0.01, 0.1, 0.5]
Batch Size	[32, 128]
Memory Length	[500, 2500, 50000]
Activation	[Tanh, ReLU]

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

Environment	Natural Gradient DQN				Baseline DQN				
	Adapt Damping	Batch Size	Memory Length	Activ-ation	Learning Rate	Exploration fraction	Batch Size	Memory Length	Activ-ation
CartPole-v0	Yes	128	500	Tanh	1e-6	0.1	32	500	Tanh
CartPole-v1	No	128	2500	Tanh	1e-6	0.01	128	50,000	Tanh
Acrobot-v1	No	128	2500	Tanh	1e-5	0.01	128	50,000	ReLU
LunarLander-v2	No	128	50,000	Tanh	1e-5	0.1	32	500	Tanh