Deep Q-Learning with Natural Gradients

Alex Barron
Department of Computer Science
Stanford University
admb@stanford.edu

Todor Markov
Department of Statistics
Stanford University
tmarkov@stanford.edu

Zack Swafford
Department of Computer Science
Stanford University
zswaff@stanford.edu

Abstract

We adapt the Natural Neural Networks proposed in Desjardins, Simonyan, Pascanu, et al. [6] to fill the role of function approximator in certain canonical deep Q-Learning problems. The natural representation of the network maintains an internal representation that creates a well-conditioned Fisher matrix, which we postulated would speed the convergence of the gradients. By using the Projected Natural Gradient Descent algorithm (PRONG), we are able to both preserve the feed-forward computation of the neural network and maintain the well-conditioned Fisher by reparametrizing and normalizing the weights of the network at each layer in a batch update. This allowed us to efficiently explore the possibility of faster and better convergence of the networks with natural gradients. In exploring the full scope of possibilities for NNNs, we found that no explicit implementations were available in the literature or online. We therefore adapted this algorithm, applied it to a problem, and included an explicit formulation below and in our associated code on GitHub [3]. If pure NNNs are not fully explored, NNNs with convolutional layers (CNNNs) are barely a footnote in some literature. We therefore adapted the general natural layer to the convolutional case, included an explicit algorithm for its computation, and made it available open source as well. We found that both the NNN and the CNNN yielded competitive results compared to typical implementations in the problems we applied them to (the CartPole problem [4] and the Gridworld [5] problem respectively). Our results show NNNs performing worse than traditional NNs on the more complex Gridworld problem but yielding comparable results on the simpler CartPole one. Without a clear result, then, NNNs and CNNNs merit further investigation.

1 Introduction

Q-learning is a model-free reinforcement learning algorithm that can be used to find or approximate the optimal action at a given state in a Markov Decision Process (MDP). The eponymous Q is a function from a state and action (s, a) to the result from taking action a in state s and following the optimal policy for the rest of the trial. The usual update rule for the Q function is based on the Bellman equation: given discount factor γ, learning rate α, and new observation $\langle s, a, r, s' \rangle$, we have

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right).$$

Given good conditions and enough observations, Q will converge to the optimal function. The optimal policy π^* in the environment can then be extracted with $\pi^*(s) = \max_a Q(s, a)$ [9].

In deep Q-Learning, a neural network is used as a function approximator to stand in for the Q function during the update step. This is particularly useful in problems with large or continuous state spaces because the Q function will be sparse even after many observations. Neural networks' primary use case is standing in as function approximators in large or continuous spaces because, when trained correctly, they build up internal representations which naturally interpolate the data and generalize to new examples. Neural networks would not be nearly as well-suited to environments with small state and action spaces or to spaces with low state and action dimensionality [16]. For example, if $s, a \in \mathbb{R}$, then the input to the NN will have dimension only two. With this few input factors neural networks are not particularly effective. However, if $s \in \mathbb{R}^n$ for a reasonable n, as in the problems we examine, the full potential of a neural network be realized.

2 Natural Gradients

The problems we examined had already been solved fairly well by both deep Q-learning and other model-free RL algorithms. However, we wanted to examine in particular the efficacy of taking the natural gradients when training the neural networks during deep Q-learning. Unlike traditional gradients, natural gradients are gradients in the direction of steepest descent on the manifold. In Euclidean spaces, this direction is the same as the direction of the traditional gradient ∇; in any Riemannian space the natural gradient ∇_N can be found by adjusting the traditional gradient by the inverse of the Fisher Information matrix (FIM) F, i.e.

$$\nabla_N = F^{-1}\nabla.$$

The parameter spaces of neural networks are Riemmanian [1], and in general following the natural gradients on a probabilistic manifold leads to faster convergence because the descent in that direction is steeper. Particularly in the case of neural networks, we can quantify the Kullback-Liebler (KL) divergence difference between iterations. This is constant when following natural gradients, meaning the speed of the descent does not slow in this space [6]. Descending on this transformed space further means that the result and performance of the stochastic gradient descent (SGD) are invariant to linear transformations of the inputs, something traditional networks cannot claim and which often proves to be a problem for them.

The reason that the natural gradients are not used in all SGD problems is that the FIM is computationally very difficult to find. Calculating F itself requires time which varies with respect to the square of the size of the parameters θ, but furthermore the inverse of the matrix must be computed. The PRONG algorithm suggests a way to reparametrize the NN to force the FIM to be approximately the identity, meaning that no additional computation is necessary to find F^{-1} or adapting the gradient. The only added cost comes in reparametrizing the network.

In order to keep the FIM the identity, we adapt the typical NN layer equation $h_i = f_i(V_i h_{i-1} + d_i)$ by adding whitening factors U_i and c_i to get

$$h_i = f_i(V_i U_{i-1}(h_{i-1} - c_{i-1}) + d_i).$$

We define c_{i-1} according to the mean of the inputs μ_{i-1}, which ensures that the normalized input is zero in expectation. The parameter U_{i-1} keeps the variance of the input Σ_{i-1} restricted to the identity, and it is obtained from the eigendecomposition of the variance $\Sigma_{i-1} = \tilde{U}_{i-1}\Lambda\tilde{U}_{i-1}^\top$. Specifically, the equations for these updates are

$$c_{i-1} \leftarrow \mu_{i-1} \quad \text{and} \quad U_{i-1} \leftarrow (\Lambda + \epsilon I)^{-\frac{1}{2}}\tilde{U}_{i-1}^\top,$$

where ϵ is a convergence hyperparameter. Further, we distinguish these parameters by defining the traditional set of parameters $\Omega \equiv \{V_0, d_0, \ldots, V_n, d_n\}$ and the new, whitening parameters $\Phi \equiv \{U_0, c_0, \ldots, U_{n-1}, c_{n-1}\} = \theta \setminus \Omega$. By ignoring the possibility of correlations between the backpropagated gradients at some level i, and ignoring off-block terms of the FIM (i.e. $F_{i,j}$, where

θ_i and θ_j are parameters from different layers) we find that the fact that $(U_i(h_{i-1} - c_{i-1}))$ has mean 0 and variance I implies that $F = F^{-1} = I$ [6].

By composing layers of this whitened, natural layer just as we would a typical layer, we create a natural neural network. The only issue is maintaining the whitening parameters Φ. As the algorithm progresses, the mean and variance of the input to a layer change, so these normalizing parameters must be updated and adjusted. Specifically, every T examples, and for each layer i, we project the input into the whitened space and find the canonical layer update parameters $W_i \equiv V_i U_{i-1}$ and $b_i \equiv d_i + W_i c_{i-1}$. In this space, we find the estimates $\hat{\mu}_{i-1}$ and $\hat{\Sigma}_{i-1}$ with a number of samples from the test examples $\mathcal{D}$. From these estimates, we update U_{i-1} and c_{i-1} as explained above and update the rest of the parameters accordingly. This completes the PRONG implementation of natural neural networks.

2.1 Convolutional Layers

Desjardins, Simonyan, Pascanu, et al. [6] mention only briefly that it is possible to whiten convolutional layers in manner similar to the way normal layers are whitened in the PRONG algorithm. We explored this space more fully and came up with a more specific set of update equations for a convolutional layer. If the input to a traditional convolutional layer h_{i-1} is a $w \times h$ matrix with m_{i-1} channels we have $h_{i-1} \in \mathbb{R}^{w \times h \times m_{i-1}}$. If f is the filter dimension, then $V_i \in \mathbb{R}^{f \times f \times m_{i-1} \times m_i}$ are the layer parameters. We assume that the padding on the convolution works such that the images are always $w \times h$. As is standard in convolutional layers, we do not include a bias term. Therefore the traditional layer calculation can be written as $h_i = \text{conv}(h_{i-1}, V_i)$. Following the pattern above, the whitened layer would have

$$h_i = \text{conv}(\text{conv}(h_{i-1}, U_{i-1}), V_i)$$

with whitening parameters $U_{i-1} \in \mathbb{R}^{1 \times 1 \times m_{i-1} \times m_{i-1}}$. Note that the filter dimension of U_{i-1} is 1–the filter for a given input and output channel whitens each pixel equivalently.

To calculate the projection matrix $W_i \in \mathbb{R}^{f \times f \times m_{i-1} \times m_i}$ (i.e. the matrix that projects the input to the whitened space) we examine the dimensions of U_{i-1} and V_i of size $m_{i-1} \times m_{i-1}$ and $m_{i-1} \times m_i$, respectively. We broadcast across the additional dimensions when doing the matrix multiplication $W_i = V_i U_{i-1}$ (intuitively, multiplying the constant filter value of each U with the corresponding $f \times f$ filter in every V). This creates the appropriate W_i for projection to the whitened space. The fact that we are allowed to whiten across feature maps like this is due to Ioffe and Szegedy [8].

With this W_i, every T iterations we take samples from $\mathcal{D}$ and project them into W_i's space as in the original PRONG algorithm. Taking the mean over the sample size and the two filter dimensions, we are left with a sample average $\hat{\mu}_{i-1} \in \mathbb{R}^{m_{i-1}}$. As before, we calculate the sample variance from this and take its eigendecomposition $\hat{\Sigma}_{i-1} = \tilde{U}_{i-1} \Lambda \tilde{U}_{i-1}^\top$. Finally, the update is, as above, $U_{i-1} \leftarrow (\Lambda + \epsilon I)^{-\frac{1}{2}} \tilde{U}_{i-1}^\top$ (which is initially size $m_{i-1} \times m_{i-1}$ but can be trivially broadcast to $1 \times 1 \times m_{i-1} \times m_{i-1}$). Updating the rest of the parameters accordingly completes the whitening of the convolutional layer.

3 MNIST

We first validate our PRONG implementation by training on the canonical MNIST handwritten digit data set [10]. We trained two simple neural networks. The first is a 3 layer fully connected network with hidden sizes of 128 and 32 where both hidden states are whitened. The second is a more complex convolutional network with two convolutional layers of kernel size 5 and output channels of 32 and 64 respectively, followed by one fully connected hidden layer of size 1024. We perform max pooling by a factor of 2 after each of convolutional layers. In the convolutional network, only the output of the first convolutional layer is whitened, since the aim of this network is to ensure that our convolutional whitened layers behave as expected. We use vanilla SGD with learning rate 0.01 as the optimizer across all trials and use a batch size of 100.

3.1 Results

We train natural and normal versions of each of the networks on the MNIST data set for 10000 steps and compare the speed to convergence and generalization accuracy, shown in figures 1 and 2.

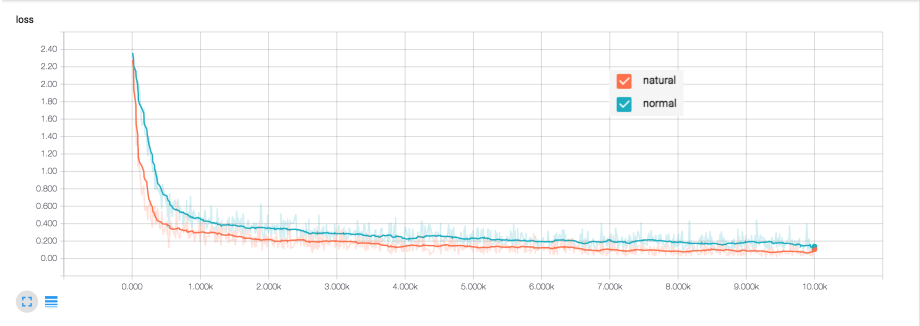

Figure 1: Fully Connected Network Training Loss

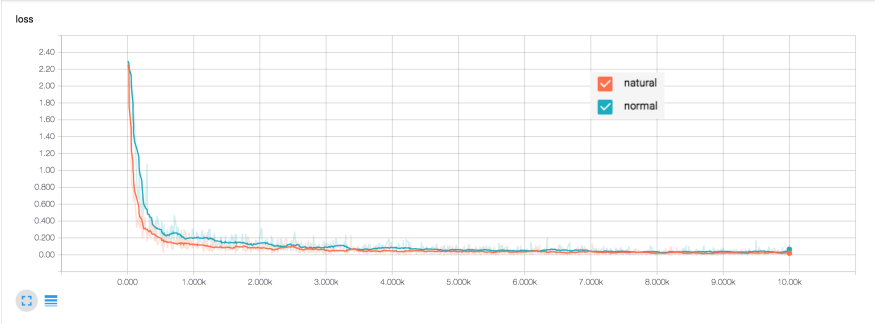

Figure 2: Convolutional Network Training Loss

In both cases the natural neural network loss (in orange) converges faster and reaches a lower final value. We also recorded the average test accuracy over 10 training runs for each network in table 1.

Gradient Type	FC Test Accuracy	Conv Test Accuracy
Natural	96.9	99.1
Vanilla SGD	95.2	98.6

Table 1: Test Accuracies

We obtain significantly higher generalization accuracy and faster training with natural gradients so it seems highly likely that we have a working PRONG implementation.

4 Cart Balancing

The CartPole problem is an MDP in a continuous space. Essentially, a cart agent is tasked with balancing a pole on top of itself. The actions are in one dimension (forward or backward) and the agent loses if the pole falls or it gets outside of certain linear boundaries. The formulation of the problem we worked with was an adaptation of the one posed by OpenAI Gym [7]. In this problem the state has $s \equiv (x, \dot{x}, \theta, \dot{\theta}) \in \mathbb{R}^4$ and the action has $a \in \mathbb{R}$. There is a reward of 1 on the last move if you win the game, and a reward of -1 on the last move if you lose; the reward is discounted by a factor $\gamma = .9$ each step. The game can end in three ways: (1) The cart goes out of bounds - i.e. runs

out of the screen (loss); (2) The pole angle goes out of bounds - i.e. the pole falls down (loss); (3) After move 200, if neither of the above happened first (win).

4.1 Results

We chose a 1 layer RELU network with experience replay as the function approximator. When the network used the PRONG algorithm, the reparametrizations were done using samples from the experience replay module (for a description of PRONG, see [6].

We evaluated the results on the CartPole environment by comparing the percentage of runs in which PRONG and SGD to a solution of the problem in 150 episodes or less. We chose this metric due to the large variance in results SGD displayed (convergence to a solution happened in anywhere between 40 to 500+ episodes, and sometimes not at all; also, the number of episodes to convergence were not normally distributed).

We ran both algorithms 20 times on the CartPole environment. Both SGD and PRONG converged to a solution 11 times out of the 20. Based on these results, we concluded that the choice of natural versus regular gradient descent has a negligible effect on convergence in the CartPole environment.

5 Gridworld

In the canonical Gridworld problem, an agent traverses a grid that has various rewards and attempt to maximize his score. Because this problem is thoroughly explored and would not be improved by NNNs, we chose a more complex formulation that first translates this Gridworld to an image, then gives this representation to the agent. Thus the state space is transformed from a simple coordinate grid to the full range of possible images of that size. In our case [2], the environment yields a 84×84 image to the agent, so the state has $s \in \mathbb{P}^{84 \times 84 \times 3}$ where $\mathbb{P}$ is the range of values a single red, green, or blue pixel value could assume. Agent's actions are the same as in the traditional Gridworld– discrete unit movement in two dimensions. There are initially six objects on the grid. Whenever the agent gets to a coordinate with an object, that object gives a reward (four objects give $+1$, two give -1; these types can be distinguished easily in the visual representation), the object is removed from the grid, and a new object with the same reward appears at random coordinates. The game progresses for exactly 200 steps with no discount factor.

5.1 Results

We implemented a 4 layer natural convolutional network as the function approximator for a Double Q learning algorithm on the Gridworld. As above in CartPole we reparametrize using samples from the experience replay module.

We were initially unable to train PRONG effectively. After extensive debugging, we traced the problem to numerical errors in Tensorflow's eigendecomposition function. The program would occasionally produce negative eigenvalues for the variance of the hidden states Σ. As $\Sigma = \mathbb{E}(hh^\top)$, Σ must be positive semidefinite and this is a computational error. We were able to fix this by implementing the PRONG+ algorithm suggested by [6], which adds batch normalization layers after each convolution and results in much more stable training.

In this setting, it seems that natural gradients were actively hurting the performance of the Q learning algorithm. Over 5 runs of both natural and normal gradients the average final reward was 12.8 for normal gradients and 0.4 for natural gradients so we were seeing very little convergence at all in the PRONG implementation.

6 Discussion

The goal of this paper was to apply natural gradient descent to deep Q-networks and evaluate its performance versus that of the regular stochastic gradient descent. The results obtained on MNIST were encouraging and strongly suggested that we have a working implementation of the projected natural gradient algorithm. Our tests with Q-learning however, were negative–using natural gradients performed comparably to SGD in the CartPole environment, and worse in Gridworld. These

results are still preliminary; we were only able to run a limited number of trials given our computational resources. It is very possible that natural neural networks require different hyperparameters to converge quickly than their regular counterparts or could even require different architectures to make the most efficient use of their alternative optimization strategy. Future work could thus perform a wide hyperparameter search to test this hypothesis and look into adapting the Q-learning algorithm to suit the natural gradient addition.

References

[1] Shun-Ichi Amari. "Natural gradient works efficiently in learning". In: *Neural computation* 10.2 (1998), pp. 251–276.

[2] Arthur Juliani (awjuliani). *Deep Reinforcement Learning Agents*. 2016. URL: https://github.com/awjuliani/DeepRL-Agents (visited on 2016-09-12).

[3] Alex Barron, Todor Markov, and Zack Swafford. *Deep Q-Learning with Natural Gradients*. 2016. URL: https://github.com/todor-markov/natural-q-learning (visited on 2016-09-12).

[4] Andrew G Barto, Richard S Sutton, and Charles W Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 834–846.

[5] Justin Boyan and Andrew W Moore. "Generalization in reinforcement learning: Safely approximating the value function". In: *Advances in neural information processing systems* (1995), pp. 369–376.

[6] Guillaume Desjardins, Karen Simonyan, Razvan Pascanu, et al. "Natural neural networks". In: *Advances in Neural Information Processing Systems*. 2015, pp. 2071–2079.

[7] OpenAI Gym. *CartPole-v0*. 2016. URL: https://gym.openai.com/envs/CartPole-v0 (visited on 2016-09-12).

[8] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[9] Mykel J Kochenderfer and Hayley J Davison Reynolds. *Decision making under uncertainty: theory and application*. MIT press, 2015.

[10] Yann LeCun, Corinna Cortes, and Christopher JC Burges. *The MNIST database of handwritten digits*. 1998.

[11] James Martens. "Deep learning via Hessian-free optimization". In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*. 2010, pp. 735–742.

[12] James Martens and Roger Grosse. "Optimizing neural networks with Kronecker-factored approximate curvature". In: *arXiv preprint arXiv:1503.05671* (2015).

[13] Yann Ollivier. "Riemannian metrics for neural networks I: feedforward networks". In: *arXiv preprint arXiv:1303.0818* (2013).

[14] Oleg Medvedev (omdv). *CartPole Tensorflow Implementation*. 2016. URL: https://gist.github.com/omdv/98351da37283c8b6161672d6d555cde6 (visited on 2016-09-12).

[15] Razvan Pascanu and Yoshua Bengio. "Revisiting natural gradient for deep networks". In: *arXiv preprint arXiv:1301.3584* (2013).

[16] Marco Wiering and Martijn Van Otterlo. "Reinforcement learning". In: *Adaptation, Learning, and Optimization* 12 (2012).