Quasi-Newton Optimization in Deep Q-Learning for Playing ATARI Games

Jacob Rafati[1] and **Roummel F. Marcia**[2]
{jrafatiheravi,rmarcia}@ucmerced.edu
[1]Electrical Engineering and Computer Science, [2]Department of Applied Mathematics
Univerisity of California, Merced
5200 North Lake Road, Merced, CA 95343, USA.

Abstract

Reinforcement Learning (RL) algorithms allow artificial agents to improve their selection of actions so as to increase rewarding experiences in their environments. The learning can become intractably slow as the state space of the environment grows. This has motivated methods that learn internal representations of the agents state by a function approximator. Impressive results have been produced by using deep artificial neural networks to learn the state representations. However, deep reinforcement learning algorithms require solving a non-convex and nonlinear unconstrained optimization problem. Methods for solving the optimization problems in deep RL are restricted to the class of first-order algorithms, like stochastic gradient descent (SGD). The major drawback of the SGD methods is that they have the undesirable effect of not escaping saddle points. Furthermore, these methods require exhaustive trial and error to fine-tune many learning parameters. Using second derivative information can result in improved convergence properties, but computing the Hessian matrix for large-scale problems is not practical. Quasi-Newton methods, like SGD, require only first-order gradient information, but they can result in superlinear convergence, which makes them attractive alternatives. The limited-memory Broyden-Fletcher-Goldfarb-Shanno (L-BFGS) approach is one of the most popular quasi-Newton methods that construct positive definite Hessian approximations. In this paper, we introduce an efficient optimization method, based on the limited memory BFGS quasi-Newton method using line search strategy – as an alternative to SGD methods. Our method bridges the disparity between first order methods and second order methods by continuing to use gradient information to calculate a low-rank Hessian approximations. We provide empirical results on variety of the classic ATARI 2600 games. Our results show a robust convergence with preferred generalization characteristics, as well as fast training time and no need for the experience replaying mechanism.

1 Introduction

Reinforcement learning (RL) a class of machine learning problems – is learning how to map situations to actions so as to maximize numerical reward signals received during the experiences that an artificial agent has as it interacts with its environment (Sutton and Barto 1998).

One of the challenges that arise real-world RL problems is "curse of dimensionality". Non-linear function approximators coupled with reinforcement learning have made it possible to learn abstractions over high dimensional state spaces (Sutton 1996; Rafati and Noelle 2015; 2017).

Successful examples of using neural networks for reinforcement learning include learning how to play the game of Backgammon at the Grand Master level (Tesauro 1995). Also, recently, researchers at DeepMind Technologies used deep convolutional neural networks (CNNs) to learn how to play some ATARI games from raw video data (Mnih et al. 2013; 2015). The resulting performance on the games was frequently at or better than the human expert level. In another effort, DeepMind used deep CNNs and a Monte Carlo Tree Search algorithm that combines supervised learning and reinforcement learning to learn how to play the game of Go at a super-human level (Silver et al. 2016).

Deep Q-learning algorithm (Mnih et al. 2013) employed a convolutional neural network (CNN) for the state-action value function $Q(s, a; w)$. They used a variant of *Temporal Difference* Learning, called *Q-Learning* to train the Deep Q-Network (DQN) to play variety of the ATARI 2600 games. Majority of deep learning problems including deep RL algorithm required solving an unconstrained optimization of a highly nonlinear and non-convex objective function of the form

$$\min_{w\in\mathbb{R}^n} \mathcal{L}(w) = \frac{1}{N}\sum_{i}^{N} \ell_i(w) \tag{1}$$

where $w \in \mathbb{R}^n$ is the vector of trainable parameters of the CNN model. There are various algorithms proposed in machine learning and optimization literature to solve (1), among those one can name first-order methods such as stochastic gradient descent (SGD) methods (Robbins and Monro 1951), and the quasi-Newton methods (Le et al. 2011). The variant of SGD method was used in DeepMind's implementation of deep Q-Learning algorithm (Mnih et al. 2013; 2015).

Since, in large-scale problems both n and N are large, the computation of the true gradient $\nabla\mathcal{L}(w)$ is expensive and the computation of the true Hessian $\nabla^2\mathcal{L}(w)$ is not practical. Hence, most of the optimization algorithms in machine learning and deep learning literature are restricted to the variant of first-order gradient descent methods such as SGD methods. SGD methods use a small random sample of data to compute an approximate of the gradient of the objective function.

The computational cost-per-iteration of SGD algorithm is small, making them the most widely used optimization method for vast majority of the deep learning and deep RL applications. However, these methods require fine-tuning of many hyperparameters, including the learning rates. The learning rates are usually chosen to be very small to decrease the undesirable effect of the noisy stochastic gradient. Therefore, deep RL methods based on the SGD algorithms require storing a large memory of the recent experiences into a *experience replay memory* $\mathcal{D}$ and replaying this memory over and over. Another major drawback of the SGD methods is that they struggle with saddle-points that occur in most of the non-convex optimization problem and has the undesirable effect on the model's generalization of learning.

In the other hand, using the second-order curvature information, can help with more robust convergence for the non-convex optimization problem. The second-order methods like Newton method use the Hessian $\nabla^2 \mathcal{L}(w)$ and the gradient to find the search direction, $p_k = -\nabla^2 \mathcal{L}(w_k)^{-1} \nabla \mathcal{L}(w_k)$ and then use line-search method to find the step length along the search direction. The main bottleneck in the second-order methods is the serious computational challenges involved in computation of the Hessian $\nabla^2 \mathcal{L}(w)$ for deep reinforcement learning problems, which is not practical when n is large. The quasi-Newton methods and Hessian-free methods are both using approaches to approximate the Hessian matrix without computing and storing the true Hessian matrix $\nabla^2 \mathcal{L}(w)$.

Quasi-Newton methods form an alternative class of first-order methods for solving the large-scale non-convex optimization problem in deep learning. These methods, like SGD, require only computing the first-order gradient of the objective function. By measuring and storing the difference between consecutive gradients, quasi-Newton methods construct *quasi-Newton matrices* $\{B_k\}$ which are low-rank updates to the previous Hessian approximations for estimating $\nabla^2 \mathcal{L}(w_k)$ at each iteration. They build a quadratic model of the objective function by using these quasi-Newton matrices and use that model to find a sequence of search directions that can result in superlinear convergence. Since these methods do not require the second-order derivatives, they are more efficient than Newton's method for large-scale optimization problems (Nocedal and Wright 2006).

There are various quasi-Newton methods proposed in literature. They differ in how they define and construct the quasi-Newton matrices $\{B_k\}$, how the search directions are computed, and how the parameters of the model are updated. The Broyden-Fletcher-Goldfarb-Shanno (BFGS) method (Broyden 1970; Fletcher 1970; Goldfarb 1970; Shanno 1970) is considered the most popular quasi-Newton algorithm, which produces positive semidefinite matrix B_k for each iteration.

The *Limited-memory* BFGS (L-BFGS) method constructs a sequence of low-rank updates to the Hessian approximation and consequently solving $p_k = B_k^{-1} \nabla \mathcal{L}(w_k)$ can be done efficiently. Recently, an L-BFGS quasi-Newton method have been implemented and employed for the classification task in the deep learning framework (Rafati, DeGuchy, and Marcia 2018; Rafati and Marcia 2018).

These methods approximate second derivative information, improving the quality of each training iteration and circumventing the need for application-specific parameter tuning. Given that the quasi-Newton methods are efficient in supervised learning problems, an important question arises: Is it also possible to use the quasi-Newton methods to learn the state representations in deep reinforcement learning successfully? We will investigate this question in this work.

In this paper, we implement a limited-memory L-BFGS optimization method, for deep reinforcement learning framework. Our deep L-BFGS Q-learning method is designed to be efficient for parallel computation in GPU. We experiment our algorithm on a variety of the ATARI 2600 games, by comparing its ability to learn robust representations of the state action value functions, as well as the computation and memory efficiency.

2 Reinforcement Learning Problem

The Reinforcement Learning (RL) problem is learning through interaction with an *environment* (Sutton and Barto 1998). At any time step the *agent* receives a representation of the environment's *state*, $s \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states, and, on that basis, the agent selects an *action*, $a \in \mathcal{A}$, where $\mathcal{A}$ is the set of all available actions. One time step later as a consequence of the agent's action, the agent receives a *reward* $r \in \mathbb{R}$ and also an update on the agent's new state, s', from the environment. Each cycle of interaction is called a transition *experience*, $e = (s, a, r, s')$. Figure 1 summarizes the agent/environment interaction.

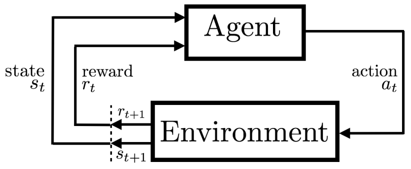


Figure 1: The agent/environment interaction in reinforcement learning.

At each time step, the agent implements a mapping from states space to actions space, $\pi : \mathcal{S} \to \mathcal{A}$, called its *policy*. The goal of the RL agent is to find an *optimal policy* that maximizes the expected value of the *return*, i.e. the cumulative sum of future rewards, $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'+1}$, where $\gamma \in (0, 1]$ is the *discounted factor* and T is a final step. The Temporal Difference (TD) learning approach is a class of *model-free* RL methods that attempt to learn a policy without learning a model of the environment. It is often useful to define a *value* function $Q_\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ to estimate the expected value of the return, following policy π. When the state space is large, or not all states are observable, we can use a function approximator $Q(s, a; w)$, such as an artificial neural network, to estimate the value function Q_π. Q-learning is a TD algorithm that attempts to find the optimal value function by minimizing the loss function $L(w)$, which is defined as the expectation of squared *TD error* over a recent transition *experience memory* $\mathcal{D}$:

$$\min_{w \in \mathbb{R}^n} \mathcal{L}(w) \triangleq \frac{1}{2}\mathbb{E}_{e\sim\mathcal{D}}\Big[\big(\mathcal{Y} - Q(s,a;w)\big)^2\Big], \tag{2}$$

where $\mathcal{Y} = r + \max_{a'} Q(s', a'; w)$ is the target value for the expected return based on the *Bellman's optimality* equations (Sutton and Barto 1998).

In practice, instead of minimization of the expected risk in (2) we can define an optimization problem for the *empirical risk* as follows

$$\min_{w \in \mathbb{R}^n} \mathcal{L}(w) \triangleq \frac{1}{2|\mathcal{D}|}\sum_{e\in\mathcal{D}}\Big[\big(\mathcal{Y} - Q(s,a;w)\big)^2\Big]. \tag{3}$$

The most common approach for solving the minimization problem (3) in literature is using a variant of stochastic gradient decent (SGD) method (3). At each optimization step k, a small set of experiences J_k are randomly sampled from the *experience replay memory* $\mathcal{D}$. This sample is used to compute an stochastic gradient of the objective function (i.e. the empirical risk) $\nabla\mathcal{L}(w)^{J_k}$ as an approximate for true gradient $\nabla\mathcal{L}(w)$

$$\nabla\mathcal{L}(w)^{(J_k)} \triangleq \frac{-1}{|J_k|}\sum_{e\in J_k}\Big[\big(\mathcal{Y} - Q(s,a;w)\big)\nabla Q\Big]. \tag{4}$$

The stochastic gradient then can be used to update the iterate w_k to w_{k+1}

$$w_{k+1} = w_k - \alpha_k \nabla\mathcal{L}(w_k)^{(J_k)}, \tag{5}$$

where α_k is the learning rate (step size).

3 Line-search L-BFGS Optimization

In this section, we briefly introduce a quasi-Newton optimization method based on the *line-search* strategy. Then we introduce the limited-memory BFGS method.

3.1 Line Search Method

Each iteration of a line search method computes a search direction p_k and then decides how far to move along that direction. The iteration is given by

$$w_{k+1} = w_k + \alpha_k p_k, \tag{6}$$

where α_k is called the step size.

Each iteration of the line search method computes search direction p_k by minimizing the quadratic model of the objective function defined by

$$p_k = \min_{p\in\mathbb{R}^n} q_k(p) \triangleq g_k^T p + \frac{1}{2}p^T B_k p, \tag{7}$$

where $g_k = \nabla\mathcal{L}(w_k)$ is the gradient of the objective function at w_k, and B_k is an approximation to Hessian $\nabla^2\mathcal{L}(w_k)$.

If B_k is a positive definite matrix, the minimizer of the quadratic function can be found as

$$p_k = -B_k^{-1} g_k. \tag{8}$$

The step size α_k is chosen to satisfy the sufficient decrease and curvature conditions, e.g. the Wolfe conditions (Nocedal and Wright 2006) given by

$$\mathcal{L}(w_k + \alpha_k p_k) \le \phi(w_k) + c_1\alpha_k \nabla\mathcal{L}_k^T p_k, \tag{9a}$$

$$\nabla\mathcal{L}(w_k + \alpha_k p_k)^T p_k \ge c_2 \nabla\mathcal{L}(w_k)^T p_k, \tag{9b}$$

with $0 < c_1 < c_2 < 1$.

3.2 Quasi-Newton Optimization Method

Methods that use $B_k = \nabla^2\mathcal{L}(w_k)$ for the Hessian in the quadratic model in (7) typically exhibit quadratic rates of convergence. However, in large-scale problems (where n and N are both large numbers), computing the true Hessian is not practical. In this case, quasi-Newton methods are viable alternatives because they exhibit super-linear convergence rates while maintaining memory and computational efficiency. Instead of the true Hessian, the quasi-Newton methods use an approximation B_k, which is updated after each step to take account of the additional knowledge gained during the step.

The quasi-Newton methods, like gradient descent methods, require only the computation of first-derivative information. They can construct a model of objective function by measuring the changes in the consecutive gradients for estimating the Hessian. The *quasi-Newton matrices* $\{B_k\}$ are required to satisfy the secant equation

$$B_{k+1}(w_{k+1} - w_k) \approx \nabla\mathcal{L}(w_{k+1}) - \nabla\mathcal{L}(w_k).$$

Typically, there are additional conditions imposed on B_{k+1}, such as symmetry (since the exact Hessian is symmetric), and a requirement that the update to obtain B_{k+1} from B_k is low rank, meaning that the Hessian approximations cannot change too much from one iteration to the next. Quasi-Newton methods vary in how this update is defined.

3.3 BFGS

Perhaps the most well-known among all of the quasi-Newton methods is the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update (Nocedal and Wright 2006; Liu and Nocedal 1989), given by

$$B_{k+1} = B_k - \frac{1}{\mathbf{s}_k^T B_k \mathbf{s}_k} B_k \mathbf{s}_k \mathbf{s}_k^T B_k + \frac{1}{\mathbf{y}_k^T \mathbf{s}_k}\mathbf{y}_k \mathbf{y}_k^T, \tag{10}$$

where $\mathbf{s}_k = w_{k+1} - w_k$ and $\mathbf{y}_k = \nabla\mathcal{L}(w_{k+1}) - \nabla\mathcal{L}(w_k)$. The matrices are defined recursively with the initial B_0 taken to be a $B_0 = \lambda_{k+1} I$, where the scalar $\lambda_{k+1} > 0$. The BFGS method generates positive-definite approximations whenever the initial approximation B_0 is positive definite and $s_k^T y_k > 0$.

3.4 Limited-memory BFGS

In practice, only the m most-recently computed pairs $\{(\mathbf{s}_k, \mathbf{y}_k)\}$ are stored, where $m \ll n$, typically $m \le 100$ for very large problems. This approach is often referred to as *limited-memory* BFGS (L-BFGS). Since we have to compute $p_k = -B_k^{-1} g_k$ at each iteration, we make use of the following recursive formula for $H_k = B_k^{-1}$:

$$H_{k+1} = \Big(I - \frac{\mathbf{y}_k \mathbf{s}_k^T}{\mathbf{y}_k^T \mathbf{s}_k}\Big) H_k \Big(I - \frac{\mathbf{s}_k \mathbf{y}_k^T}{\mathbf{y}_k \mathbf{s}_k^T}\Big) + \frac{\mathbf{y}_k \mathbf{y}_k^T}{\mathbf{y}_k \mathbf{s}_k^T}, \tag{11}$$

where $H_0 = \gamma_{k+1} I$, and common value for γ_{k+1} is usually chosen to be $\mathbf{y}_k^T \mathbf{s}_k / \mathbf{y}_k^T \mathbf{y}_k$ (Nocedal and Wright 2006; Rafati and Marcia 2018). The L-BFGS two-loop recursion algorithm given in Algorithm 1 can compute $p_k = -H_k g_k$ in $4mn$ operations (Nocedal and Wright 2006).

Algorithm 1 L-BFGS two-loop recursion.

```
$\mathbf{q} \leftarrow g_k = \nabla\mathcal{L}(w_k)$
for $i = k-1, \dots, k-m$ do
    $\alpha_i = \frac{\mathbf{s}_i^T q}{\mathbf{y}_i^T \mathbf{s}_i}$
    $\mathbf{q} \leftarrow \mathbf{q} - \alpha_i \mathbf{y}_i$
end for
$\mathbf{r} \leftarrow H_0 q$
for $i = k-1, \dots, k-m$ do
    $\beta = \frac{\mathbf{y}_i^T r}{\mathbf{y}_i^T \mathbf{s}_i}$
    $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{s}_i(\alpha_i - \beta)$
end for
return $-\mathbf{r} = -H_k g_k$
```

4 Deep L-BFGS Q Learning Method

In this section, we propose a novel algorithm for the optimization problem in deep Q-Learning framework, based on limited-memory BFGS method with line search. This algorithm is designed to be efficient for parallel computations on a single or multiple GPU(s). Also the experience memory $\mathcal{D}$ is emptied after each gradient computation, hence the algorithm needs much less RAM memory.

Inspired by (Berahas, Nocedal, and Takac 2016), we use the overlap between the consecutive multi-batch samples $\mathcal{O}_k = J_k \cap J_{k+1}$ to compute y_k as

$$\mathbf{y}_k = \nabla\mathcal{L}(w_{k+1})^{(\mathcal{O}_k)} - \nabla\mathcal{L}(w_k)^{(\mathcal{O}_k)}. \quad (12)$$

The use of overlap to compute y_k has been shown to result in more robust convergence in L-BFGS since L-BFGS uses gradient differences to update the Hessian approximations (see (Berahas, Nocedal, and Takac 2016; Erway et al. 2018)).

At each iteration of optimization we collect experiences in $\mathcal{D}$ up to batch size b and use the entire experience memory $\mathcal{D}$ as the overlap of consecutive samples O_k. For computing the gradient $g_k = \nabla\mathcal{L}(w_k)$ we use $J_k = O_{k-1} \cup O_k$:

$$\nabla\mathcal{L}(w_k)^{(J_k)} = \frac{1}{2}\Big(\nabla\mathcal{L}(w_k)^{(O_{k-1})} + \nabla\mathcal{L}(w_k)^{(O_k)}\Big). \quad (13)$$

Since $\nabla\mathcal{L}(w_k)^{(O_{k-1})}$ is already computed to obtain $\mathbf{y}_{k-1}$, we only need to compute $\nabla\mathcal{L}^{(\mathcal{O}_k)}(w_k)$

$$\nabla\mathcal{L}(w_k)^{(O_k)} = \frac{-1}{|D|}\sum_{e\in D}\Big[\big(\mathcal{Y} - Q(s,a;w_k)\big)\nabla Q\Big]. \quad (14)$$

Note that in order to obtain $\mathbf{y}_k$, we only need to compute $\nabla\mathcal{L}(w_{k+1})^{(O_k)}$ since $\nabla\mathcal{L}(w_k)^{(O_k)}$ is already computed.

The line search multi-batch L-BFGS optimization algorithm for deep Q-Leaning is provided in Algorithm 2.

5 Experiments on ATARI 2600 Games

We have performed experiments using Algorithm 2 on six ATARI 2600 games – Beam Rider, Breakout, Enduro, Q*bert, Seaquest, Space Invaders. We used OpenAI gym ATARI environments (Brockman et al. 2016) which is a wrapper on Arcade Learning Environment emulator (**?**).

Algorithm 2 Line search Multi-batch L-BFGS Optimization for Deep Q Learning.

```
Inputs: batch size $b$, L-BFGS memory $m$, exploration rate $\epsilon$
Initialize experience memory $\mathcal{D} \leftarrow \emptyset$ with capacity $b$
Initialize $w_0$, i.e. parameters of $Q(.,.;w)$ randomly
Initialize optimization iteration $k \leftarrow 0$
for episode $= 1, \dots, M$ do
    Initialize state $s \in \mathcal{S}$
    repeat for each step $t = 1, \dots, T$
        compute $Q(s,a;w_k)$
        $a \leftarrow$EPS-GREEDY$(Q(s,a;w_k), \epsilon)$
        Take action $a$
        Observe next state $s'$ and external reward $r$
        Store transition experience $e = \{s,a,r,s'\}$ to $\mathcal{D}$
        $s \leftarrow s'$
    until $s$ is terminal or intrinsic task is done
    if $|\mathcal{D}| == b$ then
        $O_k \leftarrow \mathcal{D}$
        Update $w_k$ by performing optimization step
        $\mathcal{D} \leftarrow \emptyset$
    end if
end for
========================================
Multi-batch line search L-BFGS Optimization step:
    Compute gradient $g_k^{(O_k)}$
    Compute gradient $g_k^{(J_k)} \leftarrow \frac{1}{2}g_k^{(O_k)} + \frac{1}{2}g_k^{(O_{k-1})}$
    Compute $p_k = -B_k^{-1} g_k^{(J_k)}$ using Algorithm 1
    Compute $\alpha_k$ by satisfying the Wolfe Conditions (9)
    Update iterate $w_{k+1} = w_k + \alpha_k p_k$
    $\mathbf{s}_k \leftarrow w_{k+1} - w_k$
    Compute $g_{k+1}^{(O_k)} = \nabla\mathcal{L}(w_{k+1})^{(O_k)}$
    $\mathbf{y}_k \leftarrow g_{k+1}^{(O_k)} - g_k^{(O_k)}$
    Store $\mathbf{s}_k$ to $S_k$ and $\mathbf{y}_k$ to $Y_k$ and remove oldest pairs
    $k \leftarrow k + 1$
```

These games have been used by other researchers with different learning methods, and hence they serve as benchmark environments for evaluation of deep reinforcement learning algorithms.

We used the DeepMind's Deep Q-Network (DQN) architecture (Mnih et al. 2015) as a function approximator for $Q(s,a;w)$. The same architecture was used to train the different ATARI games. The raw Atari frames, which are 210×160 pixel images with a 128 color palette, are preprocessed by first converting their RGB representation to gray-scale and then down-sampling it to a 110×84 image. The final input representation is obtained by cropping an 84×84 region of the image that roughly captures the playing area. The stack of the last 4 consecutive frames was used to produce the input of size $(4 \times 84 \times 84)$ to the Q-function. The first hidden layer of the network consists of a 32 convolutional filters of size 8×8 with stride 4, followed by a Rectified Linear Unit (ReLU) for nonlinearity. The second hidden layer consists of 64 convolutional filters of size

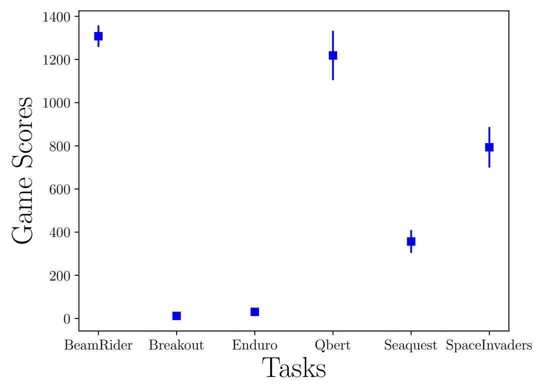


Figure 2: Test scores for six ATARI 2600 games.

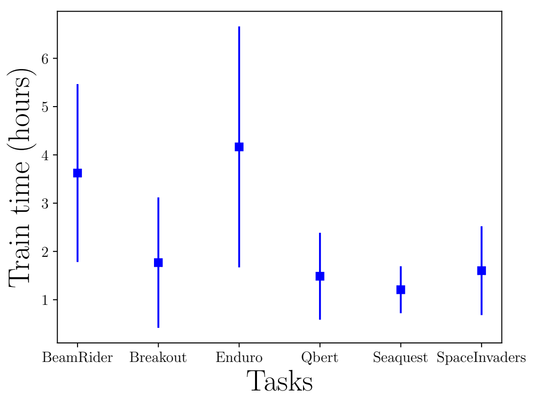


Figure 3: Total training time for six ATARI 2600 games.

4×4 with stride 2, followed by a ReLU function. The third layer consists of 512 of fully-connected linear units . followed by ReLU. The output layer is a fully-connected linear layer with a output $Q(s, a_i, w)$ for each valid joystick action $a_i \in \mathcal{A}$.

We only used 2000×1024 training steps for training the network on each game (instead of 50 million steps that was used in (Mnih et al. 2015)). The training was stopped if the norm of gradient was less than a threshold ($\|g_k\|$)

We used ϵ-greedy for exploration strategy, and similar to (Mnih et al. 2015), the exploration rare ϵ annealed linearly from 1 to 0.1. Every 10,000 steps, the performance of the learning algorithm was tested by freezing the Q-network's parameters. During the test time the greedy action $\max_a Q(s, a; w)$ was chosen by the Q-network, and there was a 5% of randomness (similar to the DeepMind's implementation in (Mnih et al. 2015)).

Inspired by (Mnih et al. 2015) we also used separate networks to compute the target values $\mathcal{Y} = r + \gamma \max_{a'} Q(s', a', w_{k-1})$, which was essentially the network with parameters in previous iterate. After each iteration of the multi-batch line search L-BFGS, w_k was updated to w_{k+1}, and the target network's parameter w_{k-1} was updated to w_k.

Our optimization method was different than DeepMind's RMSProp method (Mnih et al. 2015) (which is a variant of SGD). We used stochastic line search L-BFGS method as the optimization method (Algorithm 2). There are few important differences between our implementation of deep reinforcement learning method in comparison to the DeepMind's DQN algorithm.

We used a quite large batch size b in comparison to (Mnih et al. 2015). We experimented our algorithm with different batch sizes $b \in \{512, 1024, 2048, 4096, 8192\}$. The experience memory $\mathcal{D}$ had a capacity of b also. We used one NVIDIA Tesla K40 GPU with 12GB GDDR5 RAM. The entire experience memory $\mathcal{D}$ could fit in the GPU RAM with a batch size of $b \leq 8192$.

After every b steps of interaction with the environment, the optimization step in Algorithm 2 was ran. We used the entire $\mathcal{D}$ as the overlap O_k between two consecutive samples J_k and J_{k+1} to compute the gradient in (14) and $\mathbf{y}_k$ in (12). Although the DeepMind's algorithm is using smaller batch size of 32, but the frequency of optimization step is high (every 4 steps). We hypothesize that using the smaller batch size make the computation of the gradient too noisy, and also doesn't save much computational time, since the overhead of data transfer between GPU and CPU is more costly than the computation of the gradient on bigger batch size, due to the parallelism in GPU. Once the overlap gradient $g_k^{(O_k)}$ was computed, we could compute the gradient $g_k^{(J_k)}$ for the sample J_k from (13) by using the gradient information from the previous optimization step. Then the L-BFGS two loop-recursion in Algorithm 1 was used to compute the search direction $p_k = -H_k g_k^{(J_k)}$.

After finding the quasi-Newton decent direction, p_k, the Wolfe Condition (9) was applied to compute the step size to $\alpha_k \in [0.1, 1]$ by satisfying the sufficient decrease and the curvature conditions (Nocedal and Wright 2006). In most of the optimization steps, either the step size of $\alpha_k = 1$ satisfied the Wolfe conditions in (9), or the line search algorithm iteratively used smaller α_k until it satisfied the Wolfe conditions or reached to a lower bound of 0.1. The original DQN algorithm used a small fixed learning rate of 0.00025 to avoid the execrable drawback of the noisy stochastic gradient decent step which makes the learning to be very slow.

The vectors $\mathbf{s}_k = w_{k+1} - w_k$ and $\mathbf{y}_k = g_{k+1}^{(O_k)} - g_k^{(O_k)}$ was only added to the recent collections S_k and Y_k only if $\mathbf{s}_k^T y_k > 0$ and not close to zero. We applied this condition to preserve the positive definiteness of the L-BFGS matrices B_k. Only the m recent $\{(\mathbf{s}_i, \mathbf{y}_i)\}$ pairs were stored into S_k and Y_k ($|S_k| = m$ and $|Y_k| = m$) and the older pairs were removed from the collections. We experimented our algorithm with different L-BFGS memory sizes $m \in \{20, 40, 80\}$.

6 Results and Discussions

The average of the maximum game scores was reported in Figure 2. The error bar in Figure 2 is the standard deviation for the simulations with different batch sizes $b \in \{512, 1024, 2048, 4096\}$ and different L-BFGS mem-

Table 1: Game Scores for ATARI 2600 Games with different learning methods.

Method	Beam Rider	Breakout	Enduro	Q*bert	Seaquest	Space Invaders
Random	354	1.2	0	157	110	179
Human	7456	31	368	1952	28010	3690
Sarsa (Bellemare et al. 2013)	996	5.2	129	614	665	271
Contingency (Bellamare et al. 2012)	1743	6	159	960	723	268
HNeat Pixel (Hausknecht et al. 2014)	1332	4	91	1325	800	1145
Deep Q-Learning (Mnih et al. 2013)	4092	168	470	1952	1705	581
Offline MCTS (Guo et al. 2014)	5702	380	741	20025	2995	692
TRPO Single path (Schulman et al. 2015)	1425	10	534	1973	1908	568
TRPO Vine (Schulman et al. 2015)	859	34	431	7732	7788	450
Deep L-BFGS Q-Learning Our Method	1380	18	49	1525	600	955

ory size $m \in \{20, 40, 80\}$ for each ATARI game (total of 12 simulations per each task).

All simulations regardless of the batch size b and the memory size m performed a robust learning. The average training time for each task along with the STD error is shown in Figure 3. We did not find a correlation between the training time versus the different batch size b or the different L-BFGS memory size m. In most of the simulations, the STD error for the training time as shown in Figure 3 was not significant.

The test scores for the six ATARI 2600 environments is shown in Figure 4 (((a) Beam Rider, (b) Breakout, (c) Enduro, (d) Q*bert, (e) Seaquest, and (f) Space Invaders)) using the batch size of $b = 2048$ and L-BFGS memory size $m = 40$. The training loss ($\mathcal{L}_k$) for the six ATARI 2600 environments is shown in Figure 4 (((g) Beam Rider, (h) Breakout, (i) Enduro, (j) Q*bert, (k) Seaquest, and (l) Space Invaders)) using the batch size of $b = 2048$ and L-BFGS memory size $m = 40$.

The results of the Deep L-BFGS Q-Learning algorithm is summarized in Table 1, which also includes an expert human performance and few recent methods: Sarsa algorithm (Bellemare et al. 2013), *contingency aware* method from (Bellemare, Veness, and Bowling 2012), deep Q-learning (Mnih et al. 2013), a combination of offline Monte-Carlo Tree Search (MCTS) with supervised training from (Guo et al. 2014), and two methods based on policy optimization called Trust Region Policy Optimization (TRPO vine and TRPO single path) (Schulman et al. 2015).

The results of offline Monte-Carlo Tree Search (MCTS) method reported by (Guo et al. 2014) is generally outperforms other methods. Our learning method outperformed the other methods in Space Invaders games. Our deep L-BFGS Q-learning method consistently achieved reasonable scores in other games.

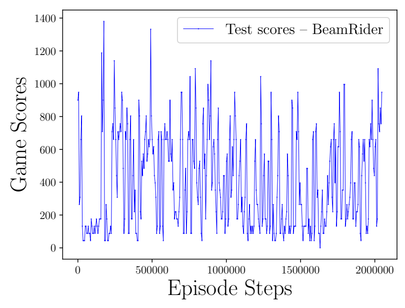
Our simulations only was trained on about 2 million steps (much less than other methods). The training time for our simulations were in the order of 3 hours, which outperformed all other methods. 500 iterations of the TRPO algorithm took about 30 hours on a 16-core computer.

7 Conclusions

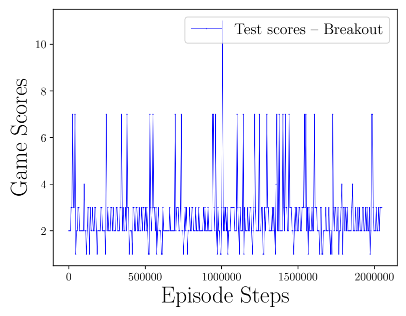
We propose and implement a novel optimization method based on line search limited-memory BFGS for deep reinforcement learning framework. We experimented our method on six classic ATARI 2600 games. The L-BFGS method attempt to approximate the Hessian matrix by constructing a positive definite low-rank matrices. Due to the non-convex and nonlinear loss function in deep reinforcement learning, our numerical experiments show that using the curvature information in computing the search direction, leads to a more robust convergence. Our deep L-BFGS Q-Learning method is designed to be efficient for parallel computations in GPU. Our method is much faster than the existing methods in the literature, and it is memory efficient since it does not need a large experience replay memory.

8 Acknowledgments

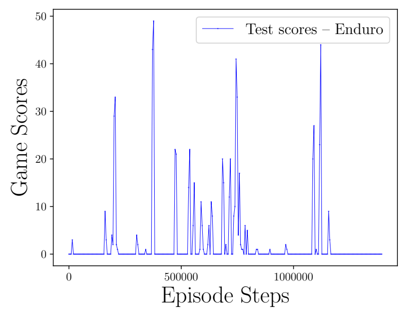
This work was supported by NSF Grant IIS-1741490. The authors gratefully acknowledge computing time on the MERCED cluster, funded by NSF Grant ACI-1429783.



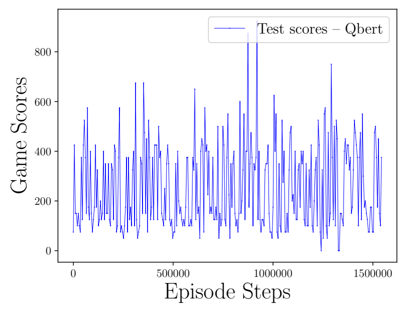
(a)



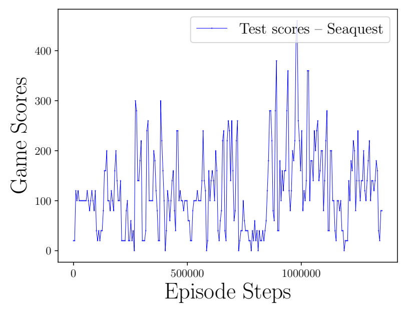
(b)



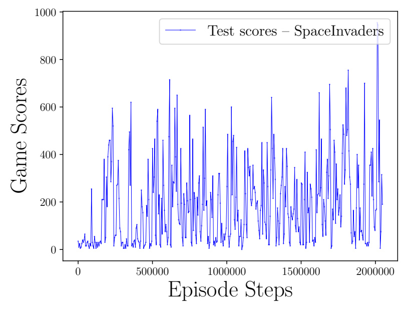
(c)



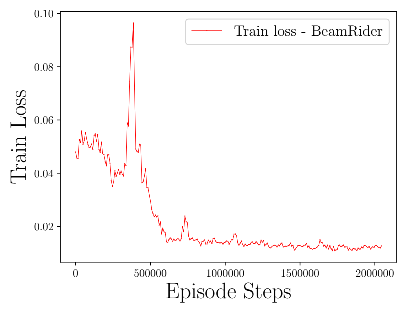
(d)



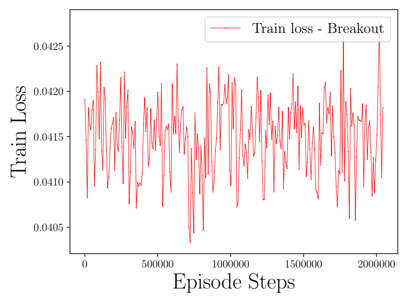
(e)



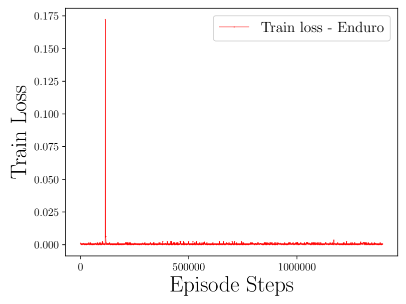
(f)



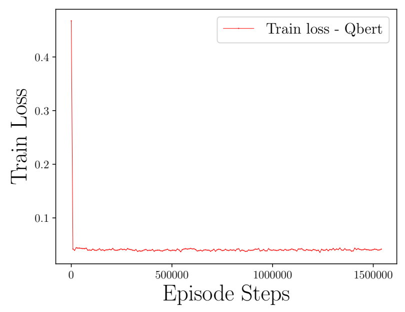
(g)



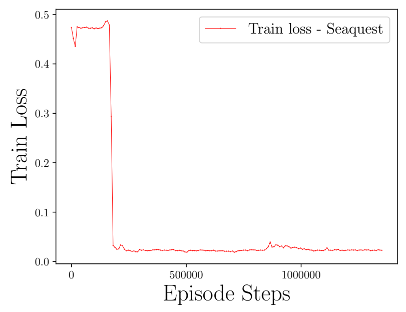
(h)



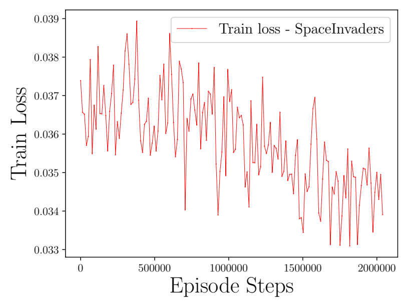
(i)



(j)



(k)



(l)

Figure 4: (a) – (f) Test scores and (g) – (l) training loss for six ATARI games. The results are form simulations with batch size $b = 2048$ and the L-BFGS memory size $m = 40$.

References

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47:253–279.

Bellemare, M.; Veness, J.; and Bowling, M. 2012. Investigating contingency awareness using atari 2600 games.

Berahas, A. S.; Nocedal, J.; and Takac, M. 2016. A multi-batch L-BFGS method for machine learning. In *Advances in Neural Information Processing Systems 29*. 1055–1063.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. Openai gym.

Broyden, C. G. 1970. The Convergence of a Class of Double-rank Minimization Algorithms 1. General Considerations. *IMA Journal of Applied Mathematics* 6(1):76–90.

Erway, J. B.; Griffin, J.; Marcia, R. F.; and Omheni, R. 2018. Trust-Region Algorithms for Training Responses: Machine Learning Methods Using Indefinite Hessian Approximations. *ArXiv e-prints*.

Fletcher, R. 1970. A new approach to variable metric algorithms. *The Computer Journal* 13(3):317–322.

Goldfarb, D. 1970. A family of variable-metric methods derived by variational means. *Mathematics of computation* 24(109):23–26.

Guo, X.; Singh, S.; Lee, H.; Lewis, R. L.; and Wang, X. 2014. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems 27*.

Hausknecht, M.; Lehman, J.; Miikkulainen, R.; and Stone, P. 2014. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games* 6(4):355–366.

Le, Q. V.; Ngiam, J.; Coates, A.; Lahiri, A.; Prochnow, B.; and Ng, A. Y. 2011. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, 265–272.

Liu, D. C., and Nocedal, J. 1989. On the limited memory BFGS method for large scale optimization. *Math. Program.* 45:503–528.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. A. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Nocedal, J., and Wright, S. J. 2006. *Numerical Optimization*. New York: Springer, 2nd edition.

Rafati, J., and Marcia, R. F. 2018. Improving l-bfgs initialization for trust-region methods in deep learning. In *17th IEEE International Conference on Machine Learning and Applications, Orlando, Florida*.

Rafati, J., and Noelle, D. C. 2015. Lateral inhibition overcomes limits of temporal difference learning. In *37th Annual Cognitive Science Society Meeting, Pasadena, CA, USA*.

Rafati, J., and Noelle, D. C. 2017. Sparse coding of learned state representations in reinforcement learning. In *Conference on Cognitive Computational Neuroscience, New York City, NY, USA*.

Rafati, J.; DeGuchy, O.; and Marcia, R. F. 2018. Trust-Region Minimization Algorithm for Training Responses (TRMinATR): The Rise of Machine Learning Techniques. In *26th European Signal Processing Conference, Rome, Italy*.

Robbins, H., and Monro, S. 1951. A stochastic approximation method. *The Annals of Mathematical Statistics* 22(3):400–407.

Schulman, J.; Levine, S.; Moritz, P.; Jordan, M.; and Abbeel, P. 2015. Trust region policy optimization. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning*.

Shanno, D. F. 1970. Conditioning of quasi-Newton methods for function minimization. *Mathematics of computation* 24(111):647–656.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489.

Sutton, R. S., and Barto, A. G. 1998. *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1st edition.

Sutton, R. S. 1996. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, 1038–1044.

Tesauro, G. 1995. Temporal difference learning and TD-Gammon. *Communications of the ACM* 38(3).