Impacts of Mathematical Optimizations on Reinforcement Learning Policy Performance

Sam Green[‡*], Craig M. Vineyard[‡], Çetin Kaya Koç[*]
[‡]Sandia National Laboratories, Albuquerque, New Mexico, USA
{sgreen, cmviney}@sandia.gov
[*]University of California Santa Barbara, Santa Barbara, California, USA
cetinkoc@ucsb.edu

***Abstract*—Deep neural networks (DNN) now outperform competing methods in many academic and industrial domains. These high-capacity universal function approximators have recently been leveraged by deep reinforcement learning (RL) algorithms to obtain impressive results for many control and decision making problems. During the past three years, research toward pruning, quantization, and compression of DNNs has reduced the mathematical, and therefore time and energy, requirements of DNN-based inference. For example, DNN optimization techniques have been developed which reduce storage requirements of VGG-16 from 552MB to 11.3MB, while maintaining the full-model accuracy for image classification. Building from DNN optimization results, the computer architecture community is taking increasing interest in exploring DNN hardware accelerator designs. Based on recent deep RL performance, we expect hardware designers to begin considering architectures appropriate for accelerating these algorithms too. However, it is currently unknown how, when, or if the "noise" introduced by DNN optimization techniques will degrade deep RL performance. This work measures these impacts, using standard OpenAI Gym benchmarks. Our results show that mathematically optimized RL policies can perform equally to full-precision RL, while requiring substantially less computation. We also observe that different optimizations are better suited than others for different problem domains. By beginning to understand the impacts of mathematical optimizations on RL policy performance, this work serves as a starting point toward the development of low power or high performance deep RL accelerators.**

I. Introduction

Reinforcement learning (RL) is a family of control techniques which arose from a combination of applying psychological models of operant conditioning with mathematical techniques of dynamic programming [26]. In RL, there is an agent that makes actions, given state observations from an environment; the environment subsequently emits rewards and new state observations, based on the agent's actions. The agent's policy π maps state observations to actions. The agent's objective is to discover an optimal policy π^*, via experience, which chooses actions that will maximize the amount of rewards it can induce from the environment. An illustration of this paradigm is shown in Fig. 1.

Formally, the goal of an agent is to find an optimal policy π^* which maximizes expected sum of rewards over some sequence of (*state*, *action*) pairs; this is represented as:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi(\tau)}\left[\sum_t r(s_t, a_t)\right] \quad (1)$$

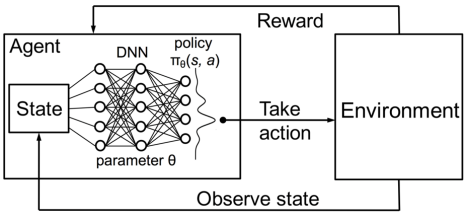

Fig. 1: Reinforcement learning computational paradigm. Based on state observations, an agent makes actions on the environment with the goal of collecting rewards.

where the *rollout* τ represents a sequence of states and actions, r is a function mapping states s and actions a to rewards, and t ranges across time steps in the rollout.

All RL techniques are devoted to solving the optimization problem given in Eq. 1. The basic methods of RL have been developed over the past several decades, but RL algorithms making use of function approximation did not work very well. However, in 2013, DeepMind utilized the high-capacity function approximation capabilities of deep neural networks (DNN) to learn a mapping from states to action selections—effectively learning a policy. By doing so, in their seminal Atari-dominating RL work, agents learned to play many Atari video games at superhuman levels of performance [19]. Since then, there has been a regular stream of RL algorithm improvements [17, 18, 23, 24], with successful application to diverse problems, including solving the game of Go [25] and robotics [3]. The application space of deep RL has naturally lagged behind theoretical results, and we expect to see further real-world results in the future. Following results, there will be increasing interest in hardware optimized for deep RL.

As hardware implementations of RL are pursued, it is only natural to build upon the optimization techniques being developed for the hardware instantiation of DNNs. Whether these techniques can be readily applied without loss of performance is unknown and is the question this research is beginning to explore. As follows, we will give an brief overview of RL, describe mathematical optimization techniques which have been applied to DNNs, and then provide our insights of the resulting impact of applying these techniques to RL policies.

978-1-5090-6014-6/18/$31.00 ©2018 IEEE

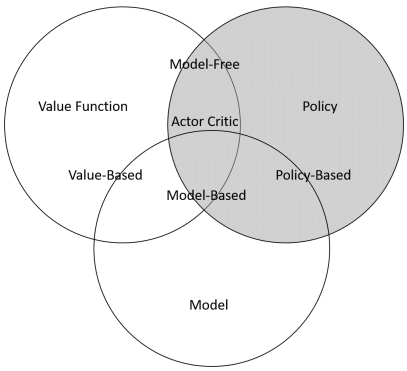

Fig. 2: Reinforcement learning taxonomy of approaches. This work is focused on a policy-based method.

TABLE I: Cost of Operations

Operation	Energy (pJ)	Area (um)
8b Add	0.03	36
16b Add	0.05	67
32b Add	0.1	137
16b FP Add	0.4	1360
32 FP Add	0.9	4184
8b Mult	0.2	282
32b Mult	3.1	3495
16b FP Mult	1.1	1640
32b FP Mult	3.7	7700
32b SRAM Read (8KB)	5	N/A
32b DRAM Read	640	N/A

Energy and die area costs for various operations. Quantized operators and operands are preferred for low-power and/or low-resource applications [28].

II. Background

A. Reinforcement Learning Approaches

The approaches for finding the optimal policy π^* in Eq. 1 may be separated into three families of methods: value-based methods, policy-based methods, and model-based methods. Value-based methods, e.g. Q-Learning, are closer to RL's historical roots in dynamic programming. They use the learned value of states and actions to find a policy which will transfer the agent into states with more value. Policy-based methods directly learn to optimize an action-making policy via maximizing a reward function. Model-based approaches require the agent have a representation of the environment which provides a prediction of the reward the environment will yield when a given action is taken. This model of the environment enables learning the optimal policy by providing a prediction of how the environment will behave so the best actions to take may be identified. As illustrated in Fig. 2, the three families of RL methods may be combined in order to benefit from the strength of each. Because they serve as the basis for deep RL, we are focusing upon policy-based methods, highlighted in gray in the figure, and described in more detail next.

"Vanilla" Policy Gradient (VPG) is a basic RL method which directly optimizes a policy function π_θ, which is represented by a DNN, parameterized by θ [29]. At each step in a Markov Decision Process, the current state observation is input to π_θ, which returns output probabilities for each possible action; the agent then samples an action from this probability distribution. Because π_θ is a differentiable function, Gradient Ascent may be used to discover local optima. As in all GA applications, an objective function is defined. In the case of GA, the objective is equal to the expected sum of rewards over a state-action sequence, and GA seeks to maximize this expectation by finding an optimal θ:

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\sum_t r(s_t, a_t)\right], \qquad (2)$$

where $\pi_\theta(\tau)$ is the distribution of state-action pairs conditioned on θ. Note that rewards are typically collected by sampling the environment with an agent using an older version of the policy, and that the policy is updated after each step of GA. Depending on the complexity of the task, the iterative process of sampling the environment, followed by GA, can take anywhere from minutes to months.

Finally, after an optimal set of parameters is found, the trained policy may be used for controlling an agent in the real-world. When applying deep RL under extreme performance constraints, the number and quantization of operations required for each DNN inference becomes a critical factor.

B. Mathematical Optimizations for Deep Learning

Modern DNN architectures require billions of floating-point multiplications and additions (MAC) for classification of a single image. Without careful design, this results in high power consumption. Gas powered vehicles, for example, can support high energy demands, but efficient, battery operated systems cannot. Additionally, modern DNNs have high latency, but low latency is required for many real-time applications. To address these challenges, a variety of numerical optimizations may be applied to DNNs.

As a motivating example, consider the DNN architecture AlexNet, which changed the machine learning landscape when it became the first DNN to dominate the ImageNet competition [13]. AlexNet requires over 724M MAC operations, using 61M 32-bit parameters, per image classification [27]. Table I gives die area and power requirements for various operations; these costs are the driving factors behind hardware and algorithmic DNN optimization efforts. At 3.7pJ per 32-bit floating-point (FP) multiplication and .9pJ per 32-bit FP addition, AlexNet costs at least 3.3mJ per inference. However, consider the results if adequate performance could be achieved using 8-bit fixed-point arithmetic: .17mJ per inference; this is a 19$\times$ reduction in power compared to 32-bit FP. Energy cost for accessing system memory is also high, with a cost of 640pJ for a single 32-bit DRAM read operation, therefore it is desirable to reduce the size of parameters such that DRAM access is minimized.

DNN mathematical optimizations are grouped by three primary strategies: pruning [15], compression [8], and quan-

tization [5]. Pruning reduces the number of neurons or the number of parameters, which, in turn, reduces the total number of MAC operations, memory storage, and traffic. Compression forces parameters to share values, thus decreasing memory storage and traffic. And quantization reduces the precision of inputs, parameters, or activations, which reduces both memory requirements and silicon required for processing elements. Some optimizations reduce power and some optimizations reduce both power and latency. It is possible to optimize a DNN and maintain classification accuracy, but there also exist extreme optimization methods which result in unavoidable accuracy loss. Depending on the application, decreased accuracy may be worth the reduction in power and latency.

Since 2015, the machine learning community has developed many pruning, quantization, and compression techniques to reduce the underlying math and data required by DNNs: [5–7, 9–12, 14, 16, 20, 21]. However, as of yet, there has been no effort toward extending these ideas to deep RL. It is necessary to investigate the exact manner in which existing DNN optimization methods impact RL performance.

III. Approach

In this work we study the effects of adapting representative pruning, quantization, and compression methods to "Vanilla" Policy Gradient (VPG). This section introduces the algorithms in the context of RL. The following section covers the results of their application to VPG.

A. Quantization

In 2015, BinaryConnect (BC) [5] was an early DNN quantization method, and exemplifies the field's approach to quantization. During forward-propagation, BC quantizes full-precision DNN parameters to $\{-1, 1\}$, using the sign function:

$$\theta_b = \begin{cases} +1 & \text{if } \theta \geq 0, \\ -1 & \text{else.} \end{cases} \tag{3}$$

Eq. 3 discards real-valued information, but, in doing so, it also eliminates the need for floating-point MACs during forward-propagation. Instead, signed floating-point addition may be used for neuron activation input calculations. During back-propagation, the error caused by quantization is used to update the real-valued θs. From a hardware perspective, when configured for AlexNet, memory overhead is $32\times$ less when using BC-derived parameters. However, there is a performance loss when using quantization; with the AlexNet topology, BinaryConnect achieves 61% top-5 accuracy on ImageNet, compared to 80.2% accuracy when using the same DNN topology and 32-bit full-precision accuracy [22].

Applied to RL, BinaryConnect may be used with Vanilla Policy Gradient. VPG minimizes the cost[1]:

$$C = -\frac{1}{T}\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t)\hat{A}_t \tag{4}$$

[1]Minimizing cost and maximizing reward are equivalent, if cost equals the negative of reward.

where $\hat{A}_t$ is the advantage at time t. Optimal calculation of $\hat{A}_t$ is a focus of RL research, but VPG sets A_t equal to the expected sum of trajectory rewards. The cost function in Eq. 4 can be combined with the BinaryConnect optimization to create the BinaryConnect+VPG method as given in Algorithm 1.

Algorithm 1 BinaryConnect+VPG

Require: A state observation, selected action, advantage, previous parameters θ_{t-1} (parameters) and b_{t-1} (biases), and learning rate η.

Ensure: Updated $\{-1, 1\}$-valued parameters θ_t and real-valued bias b_t.

1. Forward propagation:

$\theta_b \leftarrow$ binarize(θ_{t-1})

For $k = 1$ to $L - 1$, compute activation a_k, knowing a_{k-1}, θ_b and b_{t-1}

Compute output probability of selected action using softmax

2. Backward propagation:

Initialize output layer's activations gradient $\frac{\partial C}{\partial a_L}$

For $k = L$ to 2, compute $\frac{\partial C}{\partial a_{k-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and θ_b

3. Parameter update:

Compute $\frac{\partial C}{\partial \theta_b}$ and $\frac{\partial C}{\partial b_{t-1}}$ knowing $\frac{\partial C}{\partial a_k}$ and a_{k-1}

$\theta_t \rightarrow \text{clip}(\theta_{t-1} - \eta \frac{\partial C}{\partial \theta_b})$

$b_t \rightarrow b_{t-1} - \eta \frac{\partial C}{\partial b_{t-1}}$

In addition to BinaryConnect, we consider BinaryNet [10], which operates similarly, with the addition that activations are also binarized. When using BinaryNet, the activation inputs are summed, as with BinaryConnect, and then the resulting sum is converted to $[-1, 1]$ using the sign function. This optimization eliminates all full-precision calculations and replaces them with signed integer calculations. As with BinaryConnect, BinaryNet requires full-precision gradient updates during training. As an example of the impact BinaryNet quantization has on performance, it achieves 50.42% top-5 accuracy on AlexNet [22]. BinaryNet may also be combined with VPG.

B. Compression

Modern DNN architectures require over 500MB of model parameters to be transferred from memory to the processing elements [27]. Compression methods reduce the amount of data to be transferred from system memory to processing elements, thereby reducing the most expensive power operation.

Here we consider a compression method introduced by Han, et al., which clusters parameters in each layer [8]. First, a full-precision version of the network is trained using VPG. Then the n b-bit parameters of each layer are clustered into k groups using an arbitrary clustering algorithm, e.g. K-Means. Finally, the network is fine-tuned. During the fine-tuning stage, in forward-propagation, each cluster is locked to the same value. During back-propagation, the individual gradients for each cluster are summed by their respective group. The sum of the gradients are then applied to the appropriate cluster parameters.

Algorithm 2 Compression+VPG

Require: Full-precision policy network parameterized by θ_{all}, learning rate η, and number of clusters k.

Ensure: Fine-tuned network incorporating real-valued clustered parameters θ_k for each layer.

1. Full-precision training:

For each state observation perform full-precision (FP32) network evaluation. Select actions from resulting output distributions.

At episode end, update all FP32 parameters θ_{all} using standard VPG and η. Repeat until maximum performance is achieved.

2. Compression:

For each layer, cluster parameters into k groups using K-Means algorithm, resulting in θ_k.

3. Fine-tuning:

For each state observation perform network evaluation using θ_k. Select actions from resulting output distribution.

Calculate gradient as usual.

Perform modified backpropagation: in each layer, sum partial derivatives associated with respective cluster.

Update θ_k using summed partial derivatives and η. Repeat Step 3 until maximum performance is achieved.

After training, when evaluating each layer, only the cluster indices must be transmitted, resulting in a compression rate of:

$$r = \frac{nb}{n\log_2(k) + kb}. \tag{5}$$

Complete steps for combining VPG with compression are provided in Algorithm 2.

C. Pruning

Pruning is the process of eliminating neurons or parameters. This is the oldest optimization considered by our study and dates back to LeCun, et al., 1989 [15]. In this work, we prune parameters with "small" absolute values, after the policy has been trained. The method is similar to that presented in the previous section, where, initially, the full-precision network is trained. Then parameters with an absolute value less than the p^{th} percentile are set permanently to zero. Finally, the network is fine-tuned to compensate for the missing data. In [8], pruning resulted in a 9×–13× reduction in network size, while still maintaining high accuracy. See Algorithm 3 for more details.

Algorithm 3 Pruning+VPG

Require: Full-precision policy network parameterized by θ_{all}, learning rate η, and pruning threshold parameter p.

Ensure: Fine-tuned network incorporating real-valued pruned parameters θ_p for each layer.

1. Full-precision training:

For each state observation perform full-precision (FP32) network evaluation. Select actions from resulting output distribution.

Update all FP32 parameters θ_{all} using standard VPG and η. Repeat until maximum FP32 performance is achieved.

2. Pruning:

For each layer, eliminate parameters less than the layer's p^{th} percentile, resulting in θ_p.

3. Fine-tuning:

For each state observation perform network evaluation using θ_p. Select actions from resulting output distributions.

Use VPG to update θ_p. Repeat Step 3 until maximum performance is achieved.

IV. Results

To explore the impacts of mathematical optimizations on reinforcement learning policy performance, we have implemented the representative optimizations described in Section III in Python using the PyTorch environment [2]. To explore the impact of the optimizations across a variety of domains, we have used the popular benchmark suite OpenAI Gym [1, 4]. In particular, we have used the optimization methods on three discrete action-space environments: CartPole-v0, Acrobot-v1, and Atari Pong. We compare the optimized results to full-precision VPG (FP+VPG). While the CartPole-v0 and Acrobot-v1 are deterministic control problems, it has been shown that if an RL algorithm performs successfully on those, it is a good indication that it will perform well on a more difficult problem. This heuristic holds true, for example, when using the Compression+VPG optimization method, as discussed below.

The CartPole-v0 benchmark is a finite-horizon, simulated physics control challenge in which a pole is attached to an un-actuated joint and balanced vertically upon a cart. The cart moves laterally along a track, and the goal is to apply force to the cart to keep the pole balanced. The agent is provided with state observations consisting of: cart position, angle of the pole, cart velocity, and rate of change of the angle. In OpenAI Gym, the agent may apply a force of +1 or -1 to the cart at each time step, and a reward of +1 is returned at each step that the cart is balanced. The environment returns the "done" signal when the pole moves more than 15 degrees from vertical, or the cart moves more than 2.4 units from the starting position, or if the pole is kept balanced for more than 200 time steps. The environment is considered "solved" when the agent collects an average reward of 195 over 100 episodes.

Acrobot-v1 is a two-link pendulum finite-horizon environment where only the joint between links is actuated. Initially the arm is pointed down, and it must be swung up and balanced. The agent's task is to apply joint torques such that the lower link is swung up and kept balanced. The state observations include sine and cosine of the joint angles, as well as joint velocities. In OpenAI Gym the torques may be +1, 0, or -1. The environment returns -1 reward at each step and ends in failure after 500 steps or in success if the distant

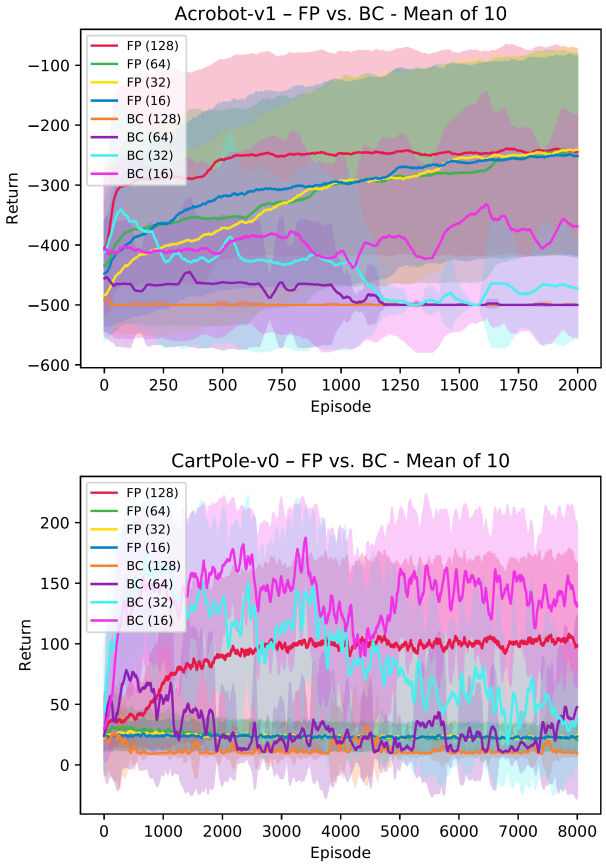

Fig. 3: (top) BinaryConnect+VPG (BC) performed poorly on the Acrobot-v1 task, compared to full-precision VPG (FP). When using 16 units in the hidden layer (the smallest version of BC) some learning takes place. (bottom) BC with 16 and 32 hidden units performs favorably compared to FP with 128 hidden units.

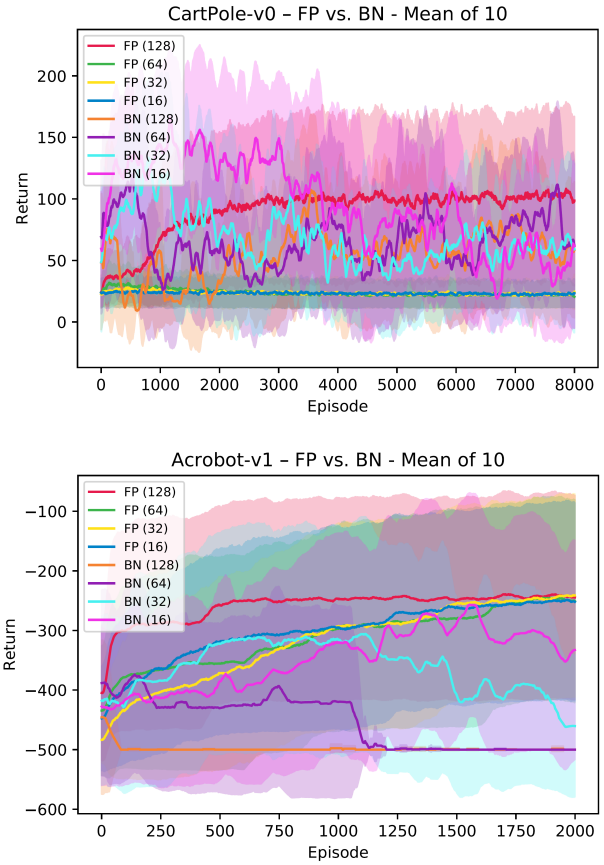

Fig. 4: (top) BinaryNet+VPG (BN) shows erratic behavior on CartPole-V0, but the 16 hidden unit version achieves continuous stretch of high returns around episodes 500–3,000, surpassing FP+VPG (FP). (bottom) BinaryNet+VPG with 16 and 32 hidden units appear to be competitive to FP+VPG results. Initial performance indicates incompatibility with the simple update method used by VPG could be the cause of eventual performance degradation.

link is elevated beyond a threshold before 500 steps.

The OpenAI Gym Atari environment is a wrapper for the Arcade Learning Environment and includes over 50 games. We learned agents for the classic Pong game, in which it competes against Pong's original AI agent. The state observation is game pixels, and the available actions are move up, move down, and no move. In OpenAI Gym, the environment terminates the game after either player reaches 21 points.

A single hidden layer neural network was selected as the base topology for all experiments. The input layer and output layer sizes varied depending on the state and action-spaces of the environment being solved. We varied the number of units in the hidden layer from 256 down to 16 for the CartPole-v0 and Acrobot-v1 tasks. For the Pong-v0 task we used 256 and 128 units in the hidden layer. In the given plots, performance is reported as the mean of ten separately trained policies. Standard deviation of each policy is also plotted.

Agent policies were initialized from the neural network topology described above, after which pruning (Pruning+VPG), quantization (BinaryConnect+VPG and BinaryNet+VPG), and compression (Compression+VPG) methods were applied as described in the algorithms above. In addition to the mathematically optimized methods, a full-precision policy (FP+VPG) was trained on each problem to provide a baseline. Agents were tasked with learning each of the previously listed environments. As can be observed in the broader RL literature, no single agent dominated all tasks. In our study we see that BinaryNet+VPG and BinaryConnect+VPG demonstrate erratic behavior on each task, with times of high and low performance, and overall they do not perform well. Pruning+VPG and Compression+VPG showed excellent performance on the control tasks. Compression+VPG dominated at the Pong task and seems to be the most generally useful of the methods considered here.

A. Impact of Quantization

BinaryConnect+VPG performed poorly on CartPole-v0, but it performed well on Acrobot-v1, as shown in Fig. 3. However, it can be observed in Fig. 3 (bottom) that BinaryConnect+VPG with 16 hidden units dominates all other variations. This may indicate that the other policies have too many parameters for this simple task. Less convincingly, as seen in Fig. 3 (top), BinaryConnect+VPG shows random spikes of marginal performance on Acrobot-v1, and it never competes with FP+VPG. On the Acrobot-v1 task, the BinaryConnect+VPG models may not have the necessary capacity to perform consistently.

In Fig. 4, it is shown that BinaryNet+VPG is a more interesting policy, with times of peak performance on both CartPole-v0 and Acrobot-v1. Its performance in Acrobot-v1 is particularly interesting and shows similar behavior to BinaryConnect+VPG, with a period of stability followed by increasing instability. Perhaps a different update strategy could prevent the instabilities.

B. Impact of Compression

Compression+VPG performed the most robustly among the mathematical optimization methods discussed in this paper. The results for Acrobot-v1, CartPole-v0, and Pong are given in Fig. 5. As described in Algorithm 2, Compression+VPG uses a policy function which has an identical topology to the full-precision version for the first half of the episodes. After the halfway point, Step 2 of Algorithm 2 is used to compress the parameters. For our experiments, k was set to 8, which limits all parameters in each layer to 8 possible 32-bit floating-point values.

During the first half of all three figures, Compression+VPG performs the same as the baseline full-precision network, as it should, because during that time it is also a full-precision network. However, after compression takes place, we see a startling reduction in variance in one case and as well as improved returns in all cases.

C. Impact of Pruning

Pruning+VPG also exhibited excellent performance on CartPole-v0, with results shown in Fig 6. As with Compression+VPG, a full-precision policy is trained during the first half of each experiment, then, as described in Algorithm 3, the lower p^{th} percentile of network parameters are eliminated. In Fig. 6, it is very promising that Pruning+VPG fully recovers after **50%** of its parameters have been removed.

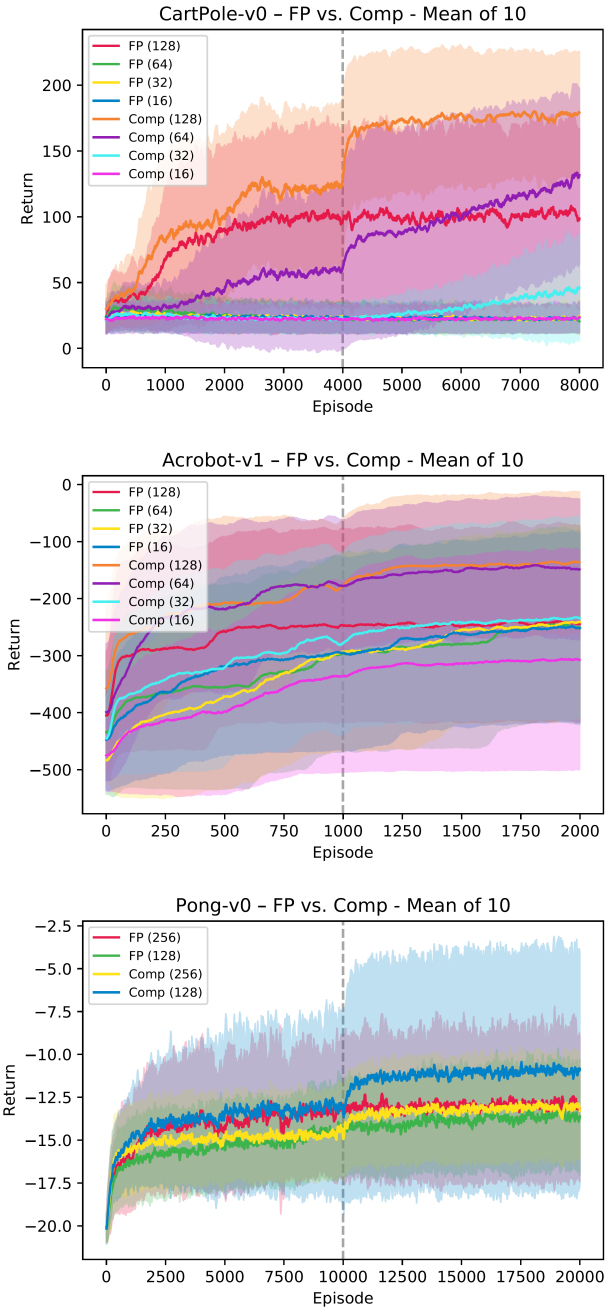

Fig. 5: (top) Compression+VPG (Comp) performs better than FP+VPG (FP) on CartPole-v0. Compression occurs at dotted line, after which performance of Compression+VPG increases. (middle) Compression+VPG performs better than FP+VPG on Acrobot-v1. Note that unlike the top and bottom plots, there is no discernible change in performance after compression for the Acrobot-v1 task. (bottom) Compression+VPG applied to Pong-v0 environment shows stronger results than FP+VPG after tuning.

V. Conclusions & Future Work

To alleviate the immense computational requirements of deep neural networks it is desirable to employ optimized versions with comparable performance by taking advantage of mathematical simplifications. A suite of such mathematical optimizations has been pursued for deep neural networks and applied to domains such as image processing. Such optimization include binarization of parameters and inputs, clustering of parameters, and pruning parameters. However, it was previously unknown whether the existing optimization techniques can be readily applied to deep RL as well, without impacting the performance of the learned policy.

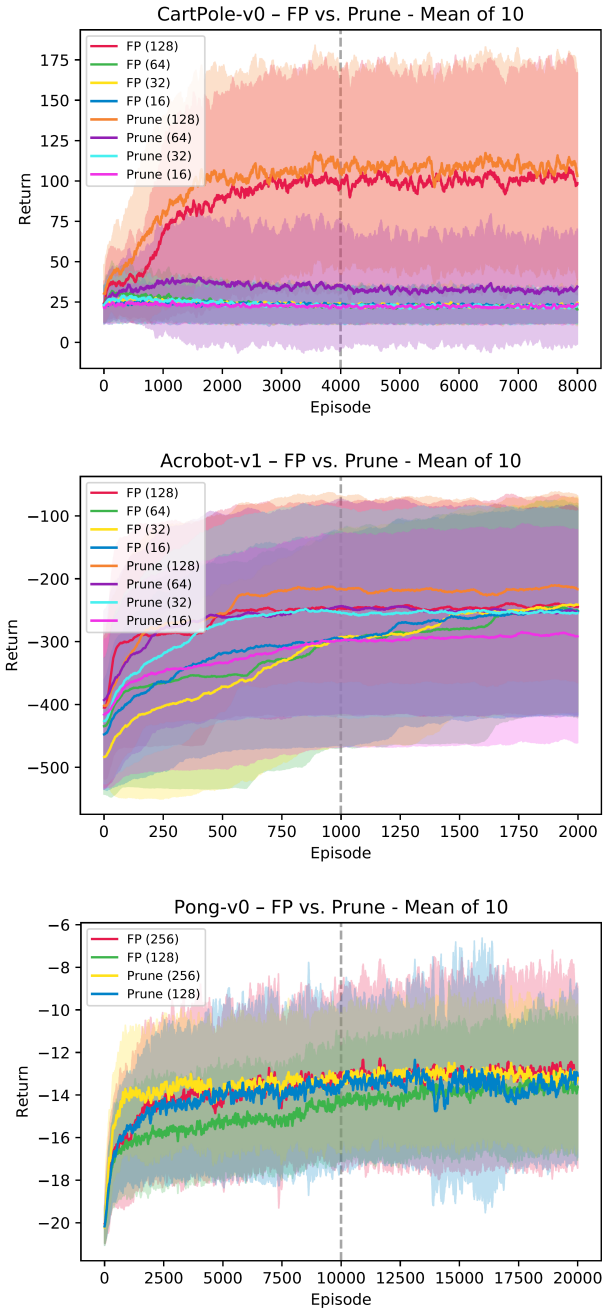

Fig. 6: Surprisingly, Pruning+VPG achieves equal performance to FP+VPG on all tasks. Policy networks are pruned at the midpoint of each plot.

In this work, we have shown initial results indicating the strong performance that may still be achieved by deep RL, even under extreme optimization. In fact, the Compression+VPG method, which locked all parameters in each layer to 8 shared values, surpassed full-precision VPG on each environment (Fig. 5). And Pruning+VPG performed equally to VPG after fine-tuning (Fig. 6). However, BinaryConnect+VPG and BinaryNet+VPG show promising, but very unstable behavior, which is most likely a result of the extreme quantization used for those methods. As is the case for reinforcement learning algorithms in general, we also observed that different optimizations are better suited than others for different problem domains. Furthermore it is still an open problem in RL to determine exactly how much model capacity is required for a particular task a priori.

VPG is a good baseline algorithm for optimized RL. It allows for experimentation with optimization methods, without confounding factors which would be included by more advanced policy-gradient based algorithms. However, VPG is notorious for exhibiting high variance, and therefore erratic collection of rewards, between policy updates. More advanced methods ensure lower variance and are also faster to train. As future work, we will explore the interactions between more sophisticated RL algorithms combined with a broader array of mathematical optimizations.

Additionally, further experiments are needed to understand the trade-offs associated with applying various optimizations to different problem domains such as continuous versus discrete action-space tasks. The neural network architecture itself is also critically important and directly affects the impact various optimizations may have. For example, an over-paramaterized neural network with more latent capacity than a given problem minimally needs will respond differently to the application of different optimization techniques than a minimal network for which optimizations may have a stronger impact. And, beyond assessing performance, more sophisticated implementation metrics can be analyzed such as: the number of multiplications and additions per policy action, size of policy parameters, and estimated power consumption.

In conclusion, the AI/ML communities have made great strides in the development of accurate and robust DNNs. The RL community is now incorporating such DNNs to an increasing degree and is showing results across a broad range of domains. Just as the architecture community has shown interest in DNN accelerator design, there will be increasing efforts toward deep RL accelerator design. However, because of the added complexity of RL, it is important to first understand the limitations of mathematical optimization for deep RL, before moving to the design of deep RL accelerators. This work shows that such a transition will be possible but future studies are required. The outcome of these studies will serve as a foundation for making architectural decisions for building RL accelerators and related neuromorphic processors.

ACKNOWLEDGMENT

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] Openai gym: A toolkit for developing and comparing reinforcement learning algorithms, 2018.

[2] Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, 2018.

[3] K. Bousmalis, A. Irpan, P. Wohlhart, Y. Bai, M. Kelcey, M. Kalakrishnan, L. Downs, J. Ibarz, P. Pastor, K. Konolige, S. Levine, and V. Vanhoucke. Using Simulation and Domain Adaptation to Improve Efficiency of Deep Robotic Grasping. *arXiv:1709.07857 [cs]*, Sept. 2017. arXiv: 1709.07857.

[4] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *arXiv:1606.01540 [cs]*, June 2016. arXiv: 1606.01540.

[5] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations. NIPS, 2015.

[6] X. Glorot, A. Bordes, and Y. Bengio. Deep Sparse Rectifier Neural Networks. PMLR, pages 315–323, June 2011.

[7] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep Learning with Limited Numerical Precision. ICML, pages 1737–1746, 2015.

[8] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. 2016.

[9] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning Both Weights and Connections for Efficient Neural Networks. NIPS, pages 1135–1143, 2015.

[10] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks. NIPS, pages 4107–4115, 2016.

[11] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *arXiv:1609.07061 [cs]*, Sept. 2016. arXiv: 1609.07061.

[12] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360 [cs]*, Feb. 2016. arXiv: 1602.07360.

[13] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[14] M. Kim and P. Smaragdis. Bitwise Neural Networks. *arXiv:1601.06071 [cs]*, Jan. 2016. arXiv: 1601.06071.

[15] Y. LeCun, J. S. Denker, and S. A. Solla. Optimal brain damage. NIPS, pages 598–605, 1990.

[16] F. Li, B. Zhang, and B. Liu. Ternary Weight Networks. *arXiv:1605.04711 [cs]*, May 2016. arXiv: 1605.04711.

[17] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv:1509.02971 [cs, stat]*, Sept. 2015. arXiv: 1509.02971.

[18] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, Feb. 2016. arXiv: 1602.01783.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

[20] J. Ott, Z. Lin, Y. Zhang, S.-C. Liu, and Y. Bengio. Recurrent Neural Networks With Limited Numerical Precision. *arXiv:1608.06902 [cs]*, Aug. 2016. arXiv: 1608.06902.

[21] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *arXiv:1603.05279 [cs]*, Mar. 2016. arXiv: 1603.05279.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[23] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. Trust Region Policy Optimization. *arXiv:1502.05477 [cs]*, Feb. 2015. arXiv: 1502.05477.

[24] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, July 2017. arXiv: 1707.06347.

[25] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, Oct. 2017.

[26] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[27] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *arXiv:1703.09039 [cs]*, Mar. 2017. arXiv: 1703.09039.

[28] D. William. High-Performance Hardware for Machine Learning. NIPS, 2015.

[29] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.