Deep Q-Networks for Accelerating the Training of Deep Neural Networks

Jie Fu* Zichuan Lin† Miao Liu‡ Nicholas Leonard§
Jiashi Feng¶ Tat-Seng Chua‖

Abstract

In this paper, we show that by feeding the weights of a deep neural network (DNN) during training into a deep Q-network (DQN) as its states, this DQN can learn policies to accelerate the training of that DNN. The actions of the DQN modify different hyperparameters during training. Empirically, this acceleration leads to better generalization performance of the DNN.

1 Introduction

The motivation for this work is founded on the proof [4] that deep networks trained by a stochastic gradient method with fewer iterations have a better generalization ability. A natural question arises: how can we accelerate the training of deep networks in practice? In this paper, we achieve this by tuning two unusual hyperparameters at every iteration: mini-batch selecting and learning rate scheduling.

The contributions of this paper are as follows:

- We propose a practical framework based on DQNs to improve modern deep neural networks' generalization performance by accelerating their training.
- We demonstrate how to use a DQN to optimize learning rates of a DNN during training. We also propose to use a DQN to select mini-batches for a DNN during training. These two new methods only require a limited "black box" interface to deep models, allowing us to tune very sophisticated, state-of-the-art deep networks without having to look under the hood.

---

*National University of Singapore, homepage: http://bigaidream.github.io/
†Tsinghua University, Beijing
‡Massachusetts Institute of Technology, homepage: http://www.mit.edu/∼miaoliu/
§Element Inc., New York City
¶National University of Singapore, homepage: https://sites.google.com/site/jshfeng/
‖National University of Singapore, homepage: http://www.comp.nus.edu.sg/∼chuats/

- A regressor is added to each layer of the deep nets whose hyperparameters are being optimized. This regressor can speed up the overall hyperparameter optimization process.

2 Related Work

A pseudo-Bayesian neural network has been used to tune a few hyperparameters of another deep neural network in [13]. The authors add a Bayesian linear regressor to the last hidden layer of a deep net, marginalizing only the output weights of the net while using a point estimate for the remaining parameters. The hyperparameters of this pesudo-Bayesian net are tuned by another Gaussian process based Bayesian optimizer.

For tuning learning rates, the most similar work is [3], where a DQN has been proposed to control one hyperparameter. However, the states require careful hand-construction of features and can only be used to tune learning rates. Furthermore, their method cannot handle stochastic environment and is thus not practical.

As for mini-batch selection, the authors in [10] study how to choose samples for a mini-batch. Their rank-based approach relies on the assumption that ranks (defined by entropies of training samples during training) of losses over samples change slowly in time, which does not hold in large datasets. Also based on the entropy of training samples, self-paced learning methods[7] are, however, usually coupled with the original optimization problems and, as far as we know, there is no work successfully applying self-paced learning to deep nets training [1].

3 Deep Q-Networks (DQNs)

The goal of a reinforcement learning agent is to maximize its expected total reward by learning an optimal policy (mapping from states to actions). At each time step t, the agent observes a state $s_t \in \mathcal{S}$ (in fact we set $s_t = w_t$), selects an action $a_t \in \mathcal{A}$, and receives a reward r_{t+1}, following the agent's decision it observes the next state s_{t+1}. The return at time t is given by $R_{t+1} = r_{t+1} + \sum_{t'=t}^{T-t-1} \gamma^{t'-t} r_t$, where T is the termination time step. In this paper, one episode is defined as one entire training of the inner loop DNN from random weights till convergence. The action-value function $Q^{\pi}(s,a) : (\mathcal{S}, \mathcal{A}) \to \mathbb{R}$ measures the expected return after observing state s_t and taking an action under a policy $\pi : \mathcal{S} \to \mathcal{A}$, $Q(s,a) = \mathbb{E}[R_t | s_t = s, a_t = a, \pi]$. The optimal action-value function $Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)$ obeys the well-known Bellman equation,

$$Q^*(s_t, a_t) = \mathbb{E}[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a]. \tag{1}$$

At each time step the estimate $\hat{y}_t$ is defined as $\hat{y}_t = Q_t(s_t, a_t)$ and the target y_t is given by $y_t = r_{t+1} + \gamma \max_{a'} Q_t(s_{t+1}, a')$. The update is based on the difference between $\hat{y}_t$ and y_t.

The DQN method [11] approximates the optimal Q-function with a DNN. We denote it as $Q(s;\theta) : \mathbb{R}^{|\mathcal{S}|} \rightarrow \mathbb{R}^{|\mathcal{A}|}$ to emphasize that this DQN is parameterized by the weights θ. It tries to minimize the expected TD error of the optimal Bellman equation: $\mathbb{E}_{s_t,a_t,r_t,s_{t+1}} \|Q_\theta(s_t, a_t) - y_t\|_2^2$, where

$$y_t = \begin{cases} r_{t+1} + \mathbf{1}_{t \neq T-1}(\gamma \max_{a'} [Q(s_{t+1}; \theta)]_{a'} & a = a_t \\ [Q(s_t; \theta)]_a & a \neq a_t \end{cases}. \tag{2}$$

DQN represents a state by a sequence of history features.

4 Generic DQN Design for Tuning Hyperparameters

4.1 Weights as States

In this paper, the states fed into a DQN are defined by the learned weights of a DNN during training. This is different from [3], where state features are manually crafted by the author. This only only avoids the troublesome feature engineering procedure, but also enables us to tune more types of hyperparameters besides learning rates. For example, as we will demonstrate later, the states are also useful for selecting mini-batches.

Because currently the best performing DNN type for supervised tasks is convolutional neural networks (CNNs), we focus on tuning the hyperparameters of CNNs. More specifically, the states are defined by the weights of those convolutional layers during training. Also because of this change[1], different from [11], we only employ a standard MLP in the DQN.

4.2 Fixing Weights by Adding Regressors

Unfortunately, due to the global update mechanism of back-propagation, the positions of filters will be changed randomly after every episode. To force the filters to appear at the same positions at every episode, we add a regressor to every layer: $\mathcal{L}_{regressor}(W_{current}, W_{last}) = \frac{1}{2} \|W_{current} - W_{last}\|^2$, where $W_{current}$ are the parameters at current iteration and W_{past} are the parameters copied from last episode after convergence.

By fixing the weights of a CNN, we have two main added benefits. The first benefit is that the repeated training processes of the CNN can be accelerated. Obviously, the training trajectories of the training of the CNN should not be significant different from each episode, if we only gradually change some hyperparameters. Therefore, this regressor will provide more information to the training of the CNN from the previous episodes.

The second byproduct is that the number of states can be decreased dramatically, which is crucial for its scability. This is due to the locality assumed by a convolutional operation, which leads to a filter at a convolutional layer being

---

[1] To some extent, we put the CNN outside the DQN.

relatively independent from each other. In other words, a small subset of filters is representative of all the filers. This is quite different from the states used by Atari games [11], where the global state representations are needed.

4.3 Reward Function

The reward functions is defined to ensure that the DQN learns a policy to find the optimal objective value in the fewest number of time steps. We define the reward function, similar to [3], as:

$$r(f, s'_t) = \frac{1}{f(s'_t) - f_{lower}}, \; f(s') > f_{lower} \, \forall s' \tag{3}$$

where f_{lower} is a predefined lower-bound constant of the training loss, s'_t is the whole weights of the CNN during training, and f is the training loss function.

5 DQN Actions for Specific Tasks

We need to determine the suitable actions for tuning learning rates and mini-batch selection.

5.1 Tuning Learning Rates

The training of a DNN with n free parameters can be formulated as the problem of minimizing a function $f : \mathbb{R}^n \to \mathbb{R}$. Following the tradition of [10, 5], we define a loss function $\psi : \mathbb{R}^n \to \mathbb{R}$ for each training sample; the distribution of training samples then induces a distribution over function $\mathcal{D}$, and the overall function f we aim to optimize is the expectation of this distribution:

$$f(w) := \mathbb{E}_{\psi \sim \mathcal{D}}[\psi(w)]. \tag{4}$$

The commonly used procedure to optimize f is to iteratively adjust w_t (the parameter vector at time step t) using gradient information obtained on a mini-batch of size b. More specifically, at each time step t and for a given $w_t \in \mathbb{R}^n$, a mini-batch $\{\psi^b_{i=1}\} \sim \mathcal{D}^b$ is selected to compute $\nabla f_t(w_t)$, where $f_t(w_t) = \frac{1}{b} \sum_{i=1}^{b} \psi_i(w_t)$.

In this work, the stochastic setting is assumed. We also assume that we are preparing training samples for a classification task. The SGD is not robust in that its performance is heavily dependent on how learning rates are tuned over time [12].

The SGD procedure then becomes a natural extension of the Gradient Descent (GD) to stochastic optimization of f as follows:

$$w_{t+1} = w_t - \eta_t \nabla f_t(w_t), \tag{5}$$

where η_t is a learning rate. It has been shown in [10] that not only the selection of the update of w_t given $\nabla f_t(w_t)$ is crucial, but the selection of the mini-batch $\{\psi_{i=1}^b\} \sim \mathcal{D}^b$ used to compute $\nabla f_t(w_t)$ also greatly contributes to the overall performance.

To tune the learning rates of the CNN, we design three actions: increasing the η_t by 0.05, decreasing η_t by 0.05 and remaining the same.

5.2 Tuning Mini-Batch Selection

Training samples are not equally valuable [9]. The next natural question is how can we squeeze the training data efficiently and effectively? Self-paced learning [8] has a similar flavor to active learning, which chooses a sample to learn from at each iteration, based on the entropy during training. Active learning approaches differ in their sample selection criteria. However, entropy is a kind of low-level information in the sense that it changes rapidly over training.

In this paper, we consider the information associate with labels as the selecting criteria, which is more close to curriculum learning [2, 7]. More specifically, DQN will decide which categories of training samples should be included in one mini-batch. Different from [7], where they encourage diverse examples within a mini-batch, we do not hard-encode such prior knowledge. Take image classification task for example, maybe in the early stage of training, it might be better to put cats and dogs into the mini-batches more frequently; but later it would be better to put dogs, cars and trucks into mini-batches more often.

The actions of the DQN are associated with distinct categories. One problem still remains: DQN can only take one action at a time. To solve this problem, we use a First-In-First-Out Queue to act as a buffer for the actions. For example if the buffer size is 5, then at every iteration the DQN will take an action and put it into the buffer. Initially, DQN has to fill the buffer with 10 actions (categories), which implies in the first 5 iterations the DQN will have no control of the CNN. After this "burn-in" phase, DQN will update this buffer with a new action.

Suppose that we are dealing with a classification task with 3 categories $\{c_1, c_2, c_3\}$, and the actions in the buffer are $[c_1, c_1, c_3, c_3, c_1]$. Then the mini-batch will consist of 60% training samples from category 1 and 40% training samples from category 3.

6 Experiments

In this section, we empirically demonstrate how a DQN can accelerate a CNN and thus improve the CNN's test accuracy. All the experiments are done on the MNIST dataset with 20,000 training samples and 10,000 testing samples. This CNN consists of 2 convolutional layers, with the first having 16 and the second having 256 filters. Above these, we add 2 fully-connected layers with 256 and 128 neurons respectively. All the non-linear functions are $Tanh$. The original learning rate is set to 0.05 in all experiments.

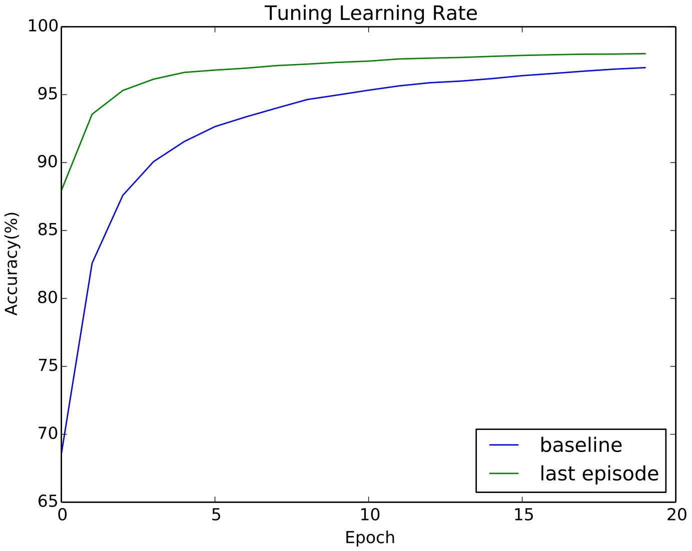

Figure 1: The testing accuracy obtained by a CNN tuned by a DQN on MNIST with 10,000 testing samples.

6.1 Tuning Learning Rates

We only test if the DQN can learn useful learning rates scheduling at all, but not intend to compare it to other adaptive learning rate schedulers.

The mini-batch size is set to 10. After training the DQN for 20 episodes, the final testing accuracy obtained by a CNN tuned by that DQN is 98.02%, whereas the baseline CNN's testing accuracy is 97%. From Figure 1 we can also observe that the convergence rate of the CNN tuned by a DQN is much faster than the baseline CNN.

6.2 Tuning Mini-Batch Selection

Here we test whether the selection of training samples makes the training algorithms converge faster.

The mini-batch size is set to 128, the buffer size is 10 and the number of epochs is 40. Actually we tried 10 and 32 as the mini-batch sizes, but the improvements were not significant. However, increasing the mini-batch size is more than a pedantic trick, as doing so could utilize the parallel computing resources more effectively [6].

As shown in Figure 2, after training the DQN for 64 episodes, the final testing accuracy of a CNN with adaptive mini-batch selected by a DQN is 90.6, whereas the baseline CNN's testing accuracy is 89.98. Figure 2 also shows that

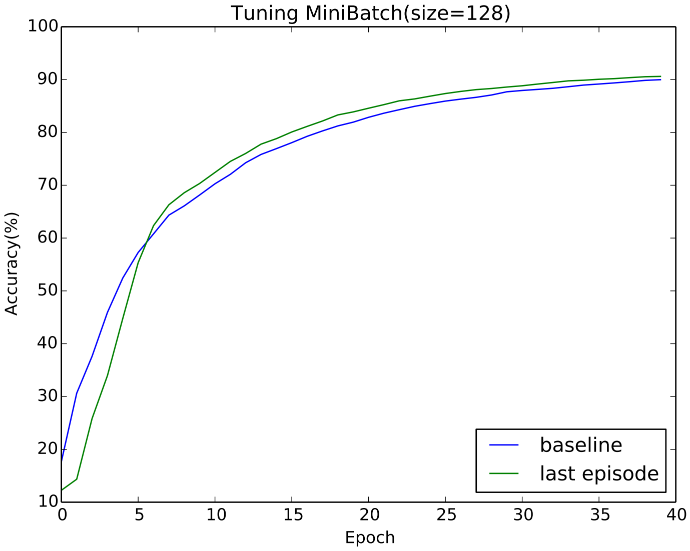

Figure 2: The testing accuracy obtained by a CNN with adaptive mini-batch selected by a DQN on MNIST with 10,000 testing samples.

a CNN with adaptive mini-batch selection performs consistantly better than the baseline CNN at every epoch.

7 Conclusion and Discussion

In this paper, we proposed a practical framework based on DQNs to improve modern deep neural networks' generalization performance by accelerating their training. We demonstrated how to use a DQN to optimize learning rates of a DNN during training. We also showed to use a DQN to select mini-batches for a DNN during training.

It has been shown that [14] lower layers features are relatively easy to transfer. Thus we are interested to study the transferrablity of the learned policies of DQNs. We would also investigate the combined effects of tuning learning rates and mini-batch selection together. Releasing an easy-to-use toolbox based on this paper is considered as one of our future works.

Acknowledgement

Jie Fu thanks Amazon Web Services for providing free GPUs. This work is also supported by NUS-Tsinghua Extreme Search (NExT) project through the National Research Foundation, Singapore.

References

[1] Vanya Avramova. Curriculum learning with deep convolutional neural networks. 2015.

[2] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[3] Samantha Hansen. Using deep q-learning to control optimization hyperparameters. *arXiv preprint arXiv:1602.04062*, 2016.

[4] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. *arXiv preprint arXiv:1509.01240*, 2015.

[5] Elad Hazan, Kfir Levy, and Shai Shalev-Shwartz. Beyond convexity: Stochastic quasi-convex optimization. In *Advances in Neural Information Processing Systems*, pages 1585–1593, 2015.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[7] Lu Jiang, Deyu Meng, Qian Zhao, Shiguang Shan, and Alexander G Hauptmann. Self-paced curriculum learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[8] M Pawan Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In *Advances in Neural Information Processing Systems*, pages 1189–1197, 2010.

[9] Agata Lapedriza, Hamed Pirsiavash, Zoya Bylinskii, and Antonio Torralba. Are all training examples equally valuable? *arXiv preprint arXiv:1311.6510*, 2013.

[10] Ilya Loshchilov and Frank Hutter. Online batch selection for faster training of neural networks. *arXiv preprint arXiv:1511.06343*, 2015.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[12] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. *arXiv preprint arXiv:1206.1106*, 2012.

[13] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md Patwary, Mostofa Ali, Ryan P Adams, et al. Scalable bayesian optimization using deep neural networks. *arXiv preprint arXiv:1502.05700*, 2015.

[14] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in Neural Information Processing Systems*, pages 3320–3328, 2014.