DEEP REINFORCEMENT LEARNING FOR ACCELERATING THE CONVERGENCE RATE

Jie Fu[ĉ*]
[ĉ] National University of Singapore, Singapore

ABSTRACT

In this paper, we propose a principled deep reinforcement learning (RL) approach that is able to accelerate the convergence rate of general deep neural networks (DNNs). With our approach, a deep RL agent (synonym for *optimizer* in this work) is used to automatically learn policies about how to schedule learning rates during the optimization of a DNN. The state features of the agent are learned from the weight statistics of the optimizee during training. The reward function of this agent is designed to learn policies that minimize the optimizee's training time given a certain performance goal. The actions of the agent correspond to changing the learning rate for the optimizee during training. As far as we know, this is the first attempt to use deep RL to learn how to optimize a large-sized DNN. We perform extensive experiments on a standard benchmark dataset and demonstrate the effectiveness of the policies learned by our approach. All source code for reproducing the experiments can be downloaded from https://github.com/bigaidream-projects/qan .

1 INTRODUCTION

Many works (Zeiler, 2012; Kingma & Ba, 2014; Schaul et al., 2012; Loshchilov & Hutter, 2016) have shown that the performance of large-sized deep models based on stochastic gradient descent (Bottou, 2010) is sensitive to learning rates. Therefore, choosing learning rates is a crucial step in the process of training deep models to achieve expected performance. For decades, the de-facto standard for selecting learning rates has been based on researchers' experiences. Recently, it has been shown that automatic hyperparameter optimization using Bayesian optimization can reach or outperform expert-level settings for deep models on a variety of benchmark datasets (Shahriari et al., 2016; Snoek et al., 2015). However, Bayesian optimization can only find one fixed learning rate to be used for all iterations in one episode [1], while changing learning rates during training usually gives better performance (Maclaurin et al., 2015). Many adaptive gradient (or more specifically adaptive learning rates) approaches have been proposed, such as Zeiler (2012); Kingma & Ba (2014); Schaul et al. (2012). However, these methods are usually designed to exploit structures in a particular domain and hardly provide good generalization performance for all datasets (Andrychowicz et al., 2016).

In this work, we propose a principled and scalable approach that enables deep reinforcement learning algorithms to control the learning rates of another large-sized DNN. With our approach, a deep RL agent (synonym for *optimizer* in this work) is used to automatically learn policies about how to schedule learning rates during the optimization of that DNN. The state features of the agent are learned from the weight statistics of the optimizee during training. The reward function of this agent is designed to learn a policy that minimize the optimizee's training time given a certain performance goal. The actions of the agent correspond to changing the learning rate for the optimizee during training. The performance of the proposed method is evaluated on a standard computer vision benchmark dataset. It only requires a light-weighted "black box" interface to deep models, allowing anybody to tune very sophisticated, state-of-the-art deep networks without having to look under the hood.

[*] http://bigaidream.github.io/contact.html

[1] In this paper, one episode is defined as one entire training process of the DNN from random weights till convergence.

Our main contributions are three-fold:

1. We propose a principled and scalable neural optimizer for automatically optimizing learning rates based on deep RL. Its success relies on the generic and compact definition of the state features based on the weight statistics of the optimizee, and the incorporation of restart into the actions taken by the RL agent.
2. We perform experiments with a state-of-the-art network architecture and gain promising results on CIFAR-10 compared to other hand-designed learning rate optimizers.
3. We observe that the greedy reward function used by the agent leads to acceptable generalization performance in practice.

2 Related Work

2.1 Hand-designed Optimizers

Various sophisticated hand-designed learning rate optitmizers have been proposed, such as Zeiler (2012); Kingma & Ba (2014); Schaul et al. (2012). Some state-of-the-art methods, such as Adam (Kingma & Ba, 2014) and ADADELTA (Zeiler, 2012), approximate the inverse Hessian, where the input to the optimizers is the diagonal of the inverse Hessian. Compared to the smooth learning rate curve given by Maclaurin et al. (2015), the ADADELTA method (Zeiler, 2012) will suggest rapidly changing learning rates.

Also, in Loshchilov & Hutter (2016), the authors show that by periodically setting the learning rate back to a large value, the convergence rate can be accelerated dramatically. However, these methods are designed to exploit structures in a particular domain and hardly provide good generalization performance for all datasets (Andrychowicz et al., 2016).

2.2 Learning to Learn

In this paper, *learning to learn* is defined as learning how to update the optimizee's parameters or hyperparameters when the optimizer is provided with error signals based on the performance of the optimizee.

Andrychowicz et al. (2016) uses an external recurrent neural network to control the weight update of a neural network, which can be seen as a special form of tuning learning rates for every weight. Unfortunately, this approach is not scalable because of the nature of recurrent neural networks, whose unrolled states must be cached for later back-propagation. The most similar work to our approach is Hansen (2016), where a deep reinforcement learning algorithm is used to control learning rates. However, in their work the agent's states require careful hand-construction of task-specific features that can only be used to tune learning rates. In contrast, our framework is more generic in that the same design can be used for tuning other hyperparameters, such as dropout ratios. Besides, the approach of Hansen (2016) cannot handle models using stochastic gradient descent whereas ours can. Another related work is Daniel et al. (2016), where they also focus on designing specific state features. In contrast, our method is based on weight statistics which is more generic and can deal SGD based optimizees. Furthermore, in our framework, the RL agent is only equipped with 2 discrete actions, i.e. restarting the learning rate to a specific value (Loshchilov & Hutter, 2016) and decreasing the learning rate by a certain amount, both of which make the exploration much more efficient.

In Snoek et al. (2015), pseudo-Bayesian neural network is used to tune hyperparameters including learning rates of another deep neural network. They add a Bayesian linear regressor to the last hidden layer of a deep network, marginalizing only the output weights of the network while using a point estimate for the remaining parameters. The hyperparameters of this pseudo-Bayesian network are tuned by a Bayesian optimizer based on Gaussian processes. This approach can only give one fixed learning rates to be used in all iterations, which is not optimal, whereas the RL agent in our work is able to output dynamic learning rates for different iterations.

In Maclaurin et al. (2015), the learning rates are modified by hypergradients. They also show that the learning rates could start by taking large values and then favor smaller values in the following iterations through a relatively smooth manner. Unfortunately, this approach is too slow to be considered

practical. Our approach only incurs negligible computational overheads for running a small-sized DL agent and thus is scalable.

3 BACKGROUND

3.1 STOCHASTIC GRADIENT DESCENT (SGD)

Training 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 convention of Loshchilov & Hutter (2015); Hazan et al. (2015), 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 to optimized is the expectation of this distribution:

$$f(w) := \mathbb{E}_{\psi \sim \mathcal{D}}[\psi(w)]. \tag{1}$$

Usually, f is optimized by iteratively adjusting w_t (the weight vector at time step t) using gradient information obtained on a mini-batch $\{\psi_{i=1}^{b}\} \sim \mathcal{D}^b$. Based on this mini-batch, the gradient $\nabla f_t(w_t)$ is computed with $f_t(w_t) = \frac{1}{b}\sum_{i=1}^{b} \psi_i(w_t)$. Then the weight vector is updated using stochastic gradient descent (SGD) as follows:

$$w_{t+1} = w_t - \eta_t \nabla f_t(w_t), \tag{2}$$

where η_t is the learning rate at time t.

3.2 DEEP REINFORCEMENT LEARNING

Reinforcement learning algorithms are designed to train an agent to interact with an environment and improve the performance in sequential decision making processes. Specifically, 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 $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}$, and receives a reward r_{t+1}. After taking the action, the agent observes the next state s_{t+1}. We assume the task of interest here is modeled by Markov Decision Processes. That is, given the current state s_t and action a_t, the probability of arriving at the next state s_{t+1} and receiving reward r_{t+1} does not depend on any of the previous states or actions. The accumulative 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, which is also called the number of episodes. 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,

$$Q^*(s_t, a_t) = \mathbb{E}[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a]. \tag{3}$$

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 (Mnih et al., 2015) approximates the optimal Q-function with a DNN. We denote it as $Q(s;\theta) : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{A}|}$ to emphasize that this DQN is parameterized by the weights θ[2]. It tries to minimize the expected Temporal Difference (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

$$y_t = \begin{cases} r_{t+1} + \mathbf{1}_{t \neq T-1}(\gamma \max_{a'} [Q(s_{t+1};\theta)]_{a'} & a = a_t \\ [Q(s_t;\theta)]_a & a \neq a_t \end{cases}. \tag{4}$$

[2]Weights θ are used to represent the weights of the DQN, whereas weights w denote the parameters of the DNN being tuned.

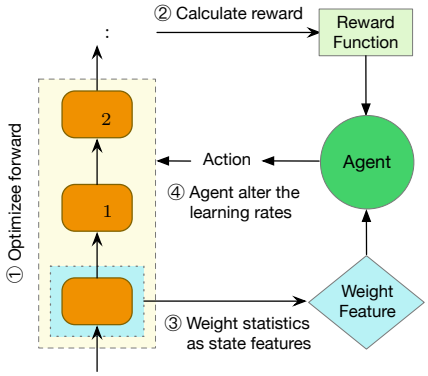

Figure 1: An overview of our architecture in one step (iteration).

4 Proposed Architecture

4.1 Overview

Figure 1 shows the procedure of training a agent to optimize the learning rates of a given optimizee network. At every iteration, we first (1) train the optimizee as usual, and then (2) obtain training loss as the reward (Section 4.3), (3) use weight statistics as state features (Section 4.2) for the agent. Finally, the agent will take one action (Section 4.4) to output a learning rate value for the optimizee.

4.2 Weight Statistics as State Features

In this paper, the state features fed into a Deep Q-Network are extracted from the weights of a DNN during training, which is different from Hansen (2016); Daniel et al. (2016), where state features are manually crafted. This kind of state representation not only avoids the troublesome feature engineering procedure, but also enables us to tune any type of hyperparameters in addition to learning rates. In this paper we only focus on tuning the learning rates of a deep model and leave other applications to future works. Our proposed method can work on any deep models without any speed-down since the dimensionality of the state is fixed for all models. Although our proposed method could work in principle for any deep models, we focus on tuning the learning rates of convolutional neural networks (CNNs).

Unfortunately, due to the global update mechanism of back-propagation, the order of convolutional filters or even the weights of those filters will be changed spontaneously across episodes. In other words, directly extracting features from optimizee weights would be too difficult. In order to make the states of the agent compact and more consistent among different episodes, we define the states based on the statistics of weights during training. Furthermore, the agent only monitors the last layer of convolutional filters to reduce the dimensionality of features.

More specifically, inspired by Domhan et al. (2015); Yamada & Morimura, we define the weight matrix w^l_{ij} in layer l that connects node i and node j. In our case, w^l_{ij} is a weight matrix of $M \times N$ elements. Also, we define $w^l = [..., w^l_{ij}, ...]$, $w^l_{i*} = [w^l_{i1}, ..., w^l_{ij}, ...]$, $w^l_{*j} = [w^l_{1j}, ..., w^l_{i*}, ...]$. The state features s_t are calculated based on $w^l(t)$, $w^l_{i*}(t)$, $w^l_{*j}(t)$, and $w^l_{ij}(t)$ of the convolutional layers at iteration t:

$$g_\alpha(w^l(t)), \tag{5}$$

$$h_\beta(\Phi^l), \Phi^l = [..., \Phi_i, ...], \Phi^l_i = g_\alpha(w^l_{i*}(t)), \tag{6}$$

$$h_\beta(\Psi^l), \Psi^l = [..., \Psi^l, ...], \Psi^l_j = g_\alpha(w^l_{*j}(t)), \tag{7}$$

$$h_\beta(\Omega^l), \Omega^l = [\Omega^l_{11}, ..., \Omega^l_{ij}, ...], \Omega^l_{ij} = g_\alpha(w^l_{ij}(t)), \tag{8}$$

where g_α with $\alpha \in \{1, ..., 12\}$ denote functions for computing mean, quantiles (0.25, 0.5, 0.75), standard deviation, skewness, kurtosis and p-th central moment ($p = 1, 2, 3, 4, 5$); h_β with $\beta \in \{1, ..., 5\}$ represent functions for computing mean, median, standard deviation, maximum value, and minimum value.

In addition to the weight statistics, the learning rate at the last iteration is also used as a feature that is similar to Mnih et al. (2014).

4.3 Reward Function

The reward function is defined to ensure that the RL agent learns a policy that finds the optimal objective value in the smallest number of iterations. Following Daniel et al. (2016), we define the reward function as:

$$r = -\frac{1}{T-1}\sum_{t=2}^{T}(log(\mathcal{L}(w_t)) - log(\mathcal{L}(w_{t-1})), \tag{9}$$

where T is the total number of iteration for an episode, and $\mathcal{L}(w_t)$ is the training loss at iteration t.

4.4 Actions

In our work, the agent has two actions: decreasing the η_t by 3%, and resetting η_t which is used in Loshchilov & Hutter (2016), in the hope that the agent could learn to generate rapidly changing learning rates. By using only 2 discrete actions, we make the exploration more efficient.

We also tried other action combinations. For example, if we provide the agent with actions of increasing the learning rate by 3%, increasing it by 10%, decreasing it by 3% and decreasing it by 10%, the agent will learn totally random actions which results in very poor training and testing performance.

5 Experiments

In this section, we empirically demonstrate how a RL agent can accelerate the convergence rate of a neural network.

5.1 Dataset and Setup

The experiments are done on the CIFAR-10 dataset (Krizhevsky & Hinton, 2009), which is a popular computer vision classification benchmark with 10 categories. We use a state-of-the-art model, wide residual network (Zagoruyko & Komodakis, 2016) as the optimizee.

In all the experiments, we set the depth to 40 convolutional layers and the wide factor to 4. The mini-batch size is 128, and the dropout ratio is 0.5 for all residual blocks. The baseline methods use exactly the same setting in Zagoruyko & Komodakis (2016) [3], where learning rate is 0.1 and learning rate decay is 0. The deep reinforcement learning agent algorithm is modified from the source code used in Mnih et al. (2015)[4], and we use the default setting to train the deep Q-network. The burn-in phase for the DQN is set to 3000 iterations, which is why the performance of DL agent shown in Figure 4 are worse than others in the beginning.

[3] `https://github.com/szagoruyko/wide-residual-networks/blob/master/scripts/train_cifar.sh`

[4] `https://github.com/kuz/DeepMind-Atari-Deep-Q-Learner`

It should be noted that the learning rates do not drop by a certain value after certain epochs (a.k.a. learning rate schedule) as it does in Zagoruyko & Komodakis (2016), and we only evaluate our methods in the first phase (the first 60 epochs). The reason is that different training phases do not share a same policy. It would be more desirable to train different agents across phases. Such settings are also used in previous "learning to learn" attempts based on neural models (Daniel et al., 2016; Hansen, 2016; Andrychowicz et al., 2016).

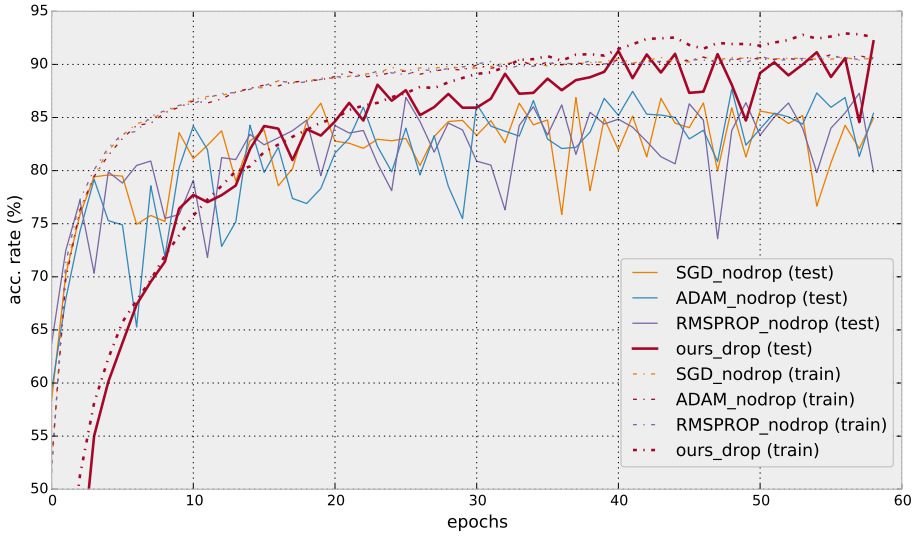

Figure 2: Results on CIFAR-10 with training and test accuracy curves by using vanilla SGD, ADAM, RMSProp and RL agent. The 'nodrop' indicates the dropout rate is set to 0, and 'drop' indicates the dropout rate is set to 0.5.

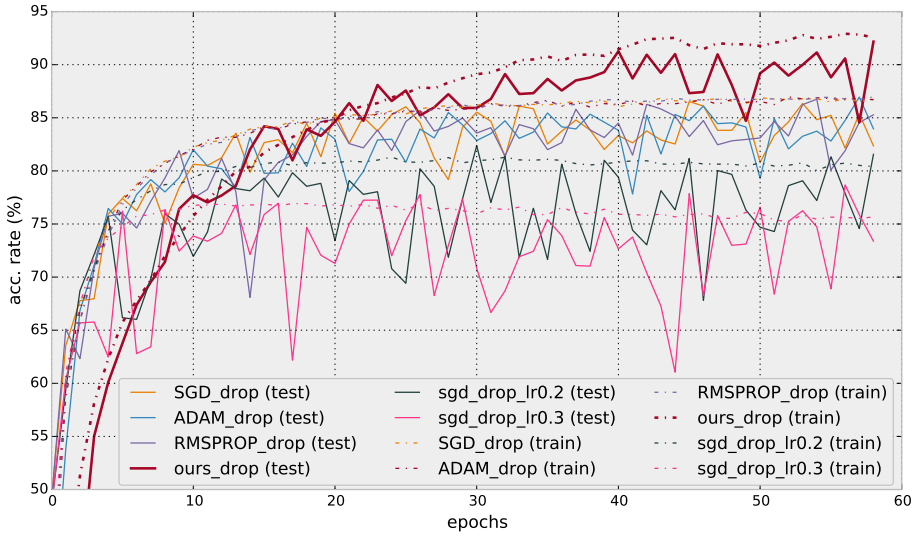

Figure 3: Results on CIFAR-10 with training and test accuracy curves by using vanilla SGD, ADAM, RMSProp and RL agent. The dropout rates of all the models are set to 0.5. "lr" indicates the initial learning rate is 0.1 (by default), 0.2,or 0.3.

5.2 RESULTS AND DISCUSSIONS

As shown in Figure 2 and Figure 3, with the help of the RL agent, the optimizee converges to better values in much shorter time than all the other adaptive and standard SGD optimizers in terms of test accuracy.

It is widely acknowledged that deep models with large number of parameters are prone to overfitting, and our proposed neural optimizer makes it even worse in the sense that our reward function implicitly encourages the optimizee to overfit the training set. As a result, we use dropout, which usually worsens the performance for wide residual networks Zagoruyko & Komodakis (2016). When training with dropout, our method still outperforms all the other optimizers (Figure 3).

Surprisingly, none of the *learning to learn* methods Loshchilov & Hutter (2016); Daniel et al. (2016); Hansen (2016); Andrychowicz et al. (2016) could learn such an abrupt or even close action like restart used in Loshchilov & Hutter (2016), which is by far the most effective hand-engineered SGD optimizer on CIFAR-10 for wide residual networks. This may imply that existing neural optimizers are still unable to learn a optimal learning rates with no prior knowledge.

Figure 4: Learning rates suggested by the RL agent for a wide residual network on CIFAR-10.

Compared to the hand-designed restart scheduler in Loshchilov & Hutter (2016), where the restart span ranges from 20 to 100 epochs, the RL agent in our work applies restart actions much more frequently as shown in Figure 4. It should be noted that the agent does learn meaningful policies, though somewhat myopic and greedy. It prefers a sequence of consecutive decreasing actions followed by a restart action. Although there is no reason to believe that such strategies are optimal, reinforcement learning with long-term predictions are inherently difficult (Li et al., 2015), especially when we are using a model-free method and each episode would take a day to play. In this work, we restrict ourselves to learning the policies from scratch with as few prior assumptions as possible, but it would be interesting to incorporate prior knowledge into the RL agent by using actor-mimic methods Parisotto et al. (2015).

It has been argued in Choromanska et al. (2014) that there are many local minima in the loss surfaces for large-sized deep models and they have similar performance on a test dataset. Based on our observations, it seems that myopic and greedy learning rate behaviors lead to faster convergence to local minima in practice. It would be interesting to empirically investigate the relationship between the size of optimizee and the behavior of their local minima found by neural optimizers.

6 CONCLUSION

In this paper, we propose to use a scalable deep reinforcement learning agent to improve modern deep neural networks' generalization performance on real-world benchmark datasets by accelerating their convergence rates. Experiments on CIFAR-10 with wide residual networks show the effectiveness of our proposed approach.

ACKNOWLEDGMENTS

We would like to thank Xipeng Qiu and Xun Huang for their helpful discussions, NVIDIA for the GPU donation and Amazon for the AWS Cloud Credits. This work is also supported by NUS-Tsinghua Extreme Search (NExT) project through the National Research Foundation, Singapore.

REFERENCES

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. *arXiv preprint arXiv:1606.04474*, 2016.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pp. 177–186. Springer, 2010.

Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surface of multilayer networks. *arXiv preprint arXiv:1412.0233*, 2014.

Christian Daniel, Jonathan Taylor, and Sebastian Nowozin. Learning step size controllers for robust neural network training. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. 2015.

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

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

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

Xiujun Li, Lihong Li, Jianfeng Gao, Xiaodong He, Jianshu Chen, Li Deng, and Ji He. Recurrent reinforcement learning: A hybrid approach. *arXiv preprint arXiv:1509.03044*, 2015.

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

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Gradient-based hyperparameter optimization through reversible learning. *arXiv preprint arXiv:1502.03492*, 2015.

Volodymyr Mnih, Nicolas Heess, Alex Graves, et al. Recurrent models of visual attention. In *Advances in Neural Information Processing Systems*, pp. 2204–2212, 2014.

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.

Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. *arXiv preprint arXiv:1511.06342*, 2015.

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

Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando de Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1): 148–175, 2016.

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.

Yasunori Yamada and Tetsuro Morimura. Weight features for predicting future model performance of deep neural networks.

Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Matthew D Zeiler. Adadelta: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.