ADAPTIVE DATASET SAMPLING BY DEEP POLICY GRADIENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Mini-batch SGD is a predominant optimization method in deep learning. Several works aim to improve naïve random dataset sampling, which appears typically in deep learning literature, with additional prior to allow faster and better performing optimization. This includes, but not limited to, importance sampling and curriculum learning. In this work, we propose an alternative way: we think of sampling as a trainable agent and let this external model learn to sample mini-batches of training set items based on the current status and recent history of the learned model. The resulting adaptive dataset sampler, named RLSampler, is a policy network implemented with simple recurrent neural networks trained by a policy gradient algorithm. We demonstrate RLSampler on image classification benchmarks with several different learner architectures and show consistent performance gain over the originally reported scores. Moreover, either a pre-sampled sequence of indices or a pre-trained RLSampler turns out to be more effective than naïve random sampling regardless of the network initialization and model architectures. Our analysis reveals the possible existence of a model-agnostic sample sequence that best represents the dataset under mini-batch SGD optimization framework.

1 INTRODUCTION

Deep learning is notoriously data-hungry. A great proportion of the recent success of deep learning algorithms has been largely attributed to the developments of bigger and better quality datasets (Russakovsky et al. (2015); Abu-El-Haija et al. (2016); Radford et al. (2019)). Commercial off-the-shelf hardware that run these algorithms has limited memory. Therefore, it is natural to partition the dataset into smaller batches of samples and train the network with a single batch at a time. The use of mini-batches under stochastic gradient descent (SGD) optimization frameworks is further justified by a regularizing effect of small batches (Wilson & Martinez (2003); Keskar et al. (2017)).

How to sample better to enhance the quality of learning has been an intriguing question from the early days of mini-batch SGD (Wilson & Martinez (2003); Bengio et al. (2009)). There have been a few works on dataset sampling strategies in various fields of machine learning (Bordes et al. (2005); Shrivastava et al. (2016); Chang et al. (2017); Katharopoulos & François (2018); Hacohen & Weinshall (2019); Dereziński et al. (2019); Ariafar et al. (2020)). Especially in reinforcement learning, where data are fed in a sequential manner, intelligent sampling techniques are proposed to produce better performance (Schaul et al. (2016); Fang et al. (2019)). However, simple random sampling without replacement is still a predominant way to sample a dataset in deep learning.

In this work, we explore a powerful alternative strategy to sample a dataset to train a deep network more effectively. Instead of considering dataset sampling as an independent and passive component of an optimization algorithm, we consider the sampling itself as a part of optimization. We model the sampling *module* as a policy network that is adaptively trained based on intermediate outputs of the learned model. This way, our sampler model, named RLSampler, can *learn* which data to be fetched next without relying on prior knowledge of the data distribution. From the learner's perspective, the only difference of this scheme is the order of a sequence of training samples fed into it. Surprisingly, deep networks trained with our RLSampler consistently outperform the ones that are trained with a naïve random sampler. Moreover, a pre-trained RLSampler or even a generated sequence used to train particular models also provides a performance boost when used for different models with different architectures and initializations.

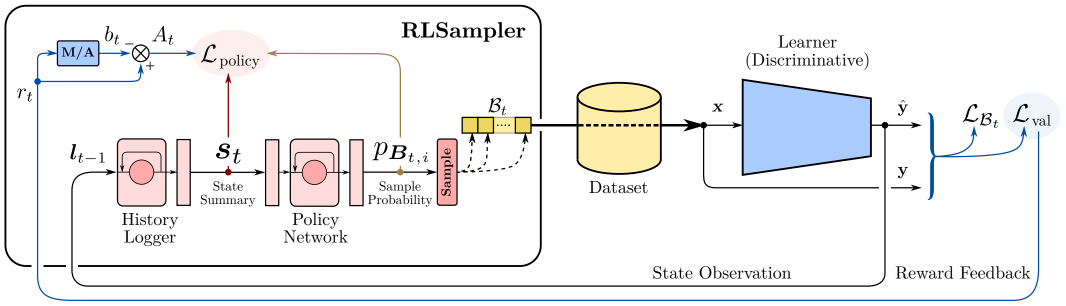


Figure 1: Training a deep discriminative model with our adaptive sampling framework (RLSampler). With two additional feedback signals, i.e., state observation and reward feedback, the dataset sampler is modeled as an RL agent that *learns* to sample optimal sequence of samples that train the learner most effectively. The interface of the learner is not affected, so the sampling method can be easily applied to any existing framework.

From those observations and subsequent analyses, we re-emphasize the role of sampling as a key to enhance the generalizability of a deep network and to stabilize its training. The remainder of the manuscript is organized as follows: In Section 2, we establish the problem of optimal sampling. We design our RLSampler in Section 3 as an RL agent trained by a policy gradient algorithm. After a brief discussion with related works in Section 4, we test our model in Section 5. We then analyze the characteristics of the generated sequences of indices produced by our RLSampler in Section 6.

2 The Problem of Optimal Sampling

A discriminative deep learning framework consists mainly of four components: a training dataset $\mathcal{X}$, a parameterized learner model, $f_\theta\colon \mathcal{X} \to \mathbb{R}^d$ with parameters θ, a scalar objective $\mathcal{L}$, and an optimization algorithm $\mathcal{E}$. In addition to those, we also consider an optionally parameterized dataset sampler S_ϕ with parameters ϕ as a separate module. The learner f_θ maps each data point $\boldsymbol{x}_i \in \mathcal{X}$ to a fixed-length logit vector $\boldsymbol{l}_i \in \mathbb{R}^d$. When the learner being trained, a mini-batch stochastic gradient-based optimization algorithm $\mathcal{E}$ tries to minimize the objective $\mathcal{L}$ with respect to the learner parameters θ. At the beginning, θ and ϕ are initialized with arbitrary values: $\theta \leftarrow \theta_0$ and $\phi \leftarrow \phi_0$. At iteration t, a sampler S_ϕ samples a mini-batch $\mathcal{B}_t$ from the dataset $\mathcal{X}$. The parameter θ is updated by the gradient of the loss, estimated from the mini-batch $\mathcal{B}_t$, i.e.,

$$\theta_{t+1} \leftarrow \theta_t - \alpha_t \nabla_\theta \mathcal{L}_{\mathcal{B}_t}(f_{\theta_t}), \tag{1}$$

where α_t is a learning rate, which is a parameter of $\mathcal{E}$. The training terminates at a finite horizon T_0, which is also a parameter of $\mathcal{E}$. Thus, our deep learning framework is a five-tuple: $(\mathcal{X}, f_\theta, \mathcal{L}, \mathcal{E}, S_\phi)$.

In the *problem of optimal sampling*, we aim to find an optimal parameter ϕ^* for the sampler S_ϕ that draws a sequence of mini-batches $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{T_0}$ of training samples which, with the algorithm $\mathcal{E}$, minimizes the generalization error of the learner network f_θ measured by $\mathcal{L}$ on a separate dataset $\mathcal{X}_{\text{test}}$. To achieve this goal, the sampler can utilize the state information of the deep learning framework. The state of the framework is a tuple $(\mathcal{X}, \mathcal{H})$ of the training set $\mathcal{X}$ and the history $\mathcal{H} \coloneqq \{\theta_{0:T_0}, \phi_{0:T_0}, \mathcal{B}_{1:T_0}\}$ of intermediate parameters and sampled indices. Our goal is to train a sampler S_ϕ like a history-dependent policy to maximize the generalizability of a trained model f_θ.

The problem can be modeled as a finite-horizon, discounted Markov Decision Process (MDP, Sutton & Barto (2018)) $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \gamma)$ and can be solved by reinforcement learning. The state space $\mathcal{S} \coloneqq (\mathcal{X}, \mathcal{H})$ is the state of the entire framework. The action space $\mathcal{A}$ is defined as a set of all indices of the dataset $\mathcal{X}$, i.e., if $N \coloneqq |\mathcal{X}|$, then $\mathcal{A} = \{1, 2, \ldots, N\}$. The state transition probability $\mathcal{P}\colon \mathcal{S} \times \mathcal{A} \to \mathbb{P}(\mathcal{S})$ is the dynamics of the deep learning framework $(\mathcal{X}, f_\theta, \mathcal{L}, \mathcal{E}, S_\phi)$. The reward $r\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ carries the information of recent generalization error of the learner model and is rigorously defined in Section 3.3. Finally, $\gamma \in (0, 1]$ is a discount factor. The optimal stochastic policy $S_\phi\colon \mathcal{S} \to \mathbb{P}(\mathcal{A})$, our RLSampler, is determined by solving the maximization problem of the expected cumulative reward,

$$J(\phi) = \mathbb{E}^{S_\phi}\left[\sum_{t=1}^{T_0} \gamma^t r(s_t, a_t)\right]. \tag{2}$$

In this work, we utilize a policy gradient algorithm (Williams (1992)) to solve the problem online.

3 Adaptive Sampler Design

3.1 Design Concerns

There are three constraints need be considered before resolving the structure of our RLSampler. The first and the most important one is an appropriate approximation of the state space $\mathcal{S}$. As mentioned in Section 2, $\mathcal{S}$ is the complete state $(\mathcal{X}, \mathcal{H})$ of the framework, which is intractable in practice. Especially, the history $\mathcal{H}$ grows as learning progresses, and its final length T_0 is almost always too large for typical hardware to process at once. In order to exploit the state information in an efficient way, we need a compact summary of it. Therefore, a sequential model is employed to summarize the history of varying length into a fixed-length vector. Furthermore, each element $(\theta_t, \phi_t, \mathcal{B}_t) \in \mathcal{H}$ of the history at time t is already a high-dimensional variable. Fortunately, we can simply ignore ϕ_t since it is a state of the sampler; however, the remaining elements $\mathcal{X}$, $\theta_{0:t-1}$, and $\mathcal{B}_{0:t-1}$ of the state $\mathcal{S}$ still require a tractable approximation. In the simplest form, a deep model f_θ can be viewed as a black-box function, the state of which can be approximated with pairs of inputs and corresponding outputs. The information of the inputs can be thought of as being already encoded in the hidden states of the sampler, as it *generates* those input indices. These inputs also represent partial information of the dataset $\mathcal{X}$ as well as the history of sampled batches $\mathcal{B}_{0:t-1}$. Thus, by feeding *only* the learner outputs to the sampler, we can provide a simple approximation of the current state $\{\mathcal{X}, \theta_{0:t-1}, \phi_{0:t-1}, \mathcal{B}_{1:t-1}\}$ of the deep learning framework. To this end, we design our RLSampler with a recurrent network that receives the output logits $\boldsymbol{l}_{t-1} = (\boldsymbol{l}_{t-1,1}, \boldsymbol{l}_{t-1,2}, \ldots, \boldsymbol{l}_{t-1,|\mathcal{B}|})$ of the learner $f_{\theta_{t-1}}$ at the previous iteration $t-1$ to sample a mini-batch $\mathcal{B}_t$ for the iteration t.

Action space $\mathcal{A}$, or the output space of the sampler S_ϕ is another important thing to be considered. Two options are the most natural for a sampler: the set of all mini-batches and the set of single indices. However, the number of possible combinations for a mini-batch becomes so large in practical deep learning frameworks that the first option is in fact infeasible. A more reasonable choice would be a set of all indices of the training set, i.e., we draw a single item for each call to the sampler, which yields the probabilities of the indices to be sampled next. With a mini-batch SGD, however, the state of the learner changes every time a *mini-batch* is sampled, rather than a single item. Thus, we split the sampler into two modules: the *history logging network*, which logs the history of the logits from the learner every mini-batch, and the *policy network*, from which items are sampled one-by-one. They will be explained in more detail in Section 3.2

The final consideration is about the reward feedback. The rewards should be evaluated on a separate validation set to provide hints on the current generalization gap. We need to partition the original training set $\mathcal{X}$ into a *training subset* $\mathcal{X}_{\text{t}}$ which is used to update the learner network, and a *validation subset* $\mathcal{X}_{\text{v}}$ which is for updating the sampler. Therefore, paired with RLSampler, the learner effectively experiences a *smaller* training set. In addition, the evaluation on the validation set is an excessive overhead, so the number of tries is best to be minimized. Also, the reward signal should properly reflect the effect of changes the policy has made. For those reasons, the sampler model need be updated less often than the learner network. Therefore we introduce two new hyperparameters: a policy update period T_1, and a reward period T_2 that divides T_1.

3.2 Controller Design

Before training, the original dataset $\mathcal{X}$ is partitioned into a training subset $\mathcal{X}_{\text{t}}$ and a validation subset $\mathcal{X}_{\text{v}}$. We refer to these splits as *subsets* to avoid unnecessary confusion with the training set $\mathcal{X}$ and the test set $\mathcal{X}_{\text{test}}$. For the same reason, we count the iteration t only with respect to the learner's update, i.e., in the t-th iteration, the learner sees t-th mini-batch from the training subset.

As in Figure 1 and Algorithm 1, we first describe how RLSampler can be used in a mini-batch SGD optimization framework. RLSampler S_ϕ is implemented as an iterator; every call to it returns a single index of the dataset. At iteration t, the sampler is called $|\mathcal{B}_t| = k$ times to fetch a mini-batch $\mathcal{B}_t = \{\boldsymbol{b}_{t,i}\}_{i=1}^{k}$ from $\mathcal{X}_{\text{t}}$. The RLSampler S_ϕ has two inputs: First, at the end of each iteration t, the RLSampler receives the learner's output logits $\boldsymbol{l}_t$ as a state observation to update the historical summary $\boldsymbol{s} \leftarrow \boldsymbol{s}_t$. Second, to train itself, every reward period T_2 of iterations, the sampler fetches the entire validation subset $\mathcal{X}_{\text{v}}$ and receives an averaged validation loss from the learner, from which a reward signal is calculated. After T_1 iterations, with T_1/T_2 reward signals received, RLSampler is trained for a single step, then all the hidden states of the recurrent modules are reset.

Algorithm 1 Mini-batch SGD with adaptive dataset sampling using an RLSampler.

```
Require: $T_0$, total training steps. $T_1$, a policy update period. $T_2$, a reward period, divides $T_1$. $k$, a batch size.
Require: $\mathcal{X} = (\mathcal{X}_\text{t}, \mathcal{X}_\text{v})$, the training set. $\theta_0$, $\phi_0$, initial parameters of the learner $f_\theta$ and the sampler $S_\phi$.
 1: for $t = 1, \cdots, T_0$ do
 2:     for $\boldsymbol{b}_{t,i}$ in the $t$-th mini-batch $\mathcal{B}_t = \{\boldsymbol{b}_{t,i}\}_{i=1}^k$ do
 3:         Using equation (4), sample $\boldsymbol{b}_{t,i} \sim \boldsymbol{B}_{t,i}$.
 4:     end for
 5:     Using equation (1), update the learner $f_{\theta_t}$ with the mini-batch $\mathcal{B}_t$.
 6:     Using equation (3), calculate a new summary $\boldsymbol{s}_{t+1}$ from the learner's output logits $\boldsymbol{l}_t = f_{\theta_t}(\mathcal{B}_t)$.
 7:     if $t$ mod $T_2 = 0$ then
 8:         Using $\mathcal{L}_\text{val} = \mathbb{E}_{(\boldsymbol{x},y)\in\mathcal{X}_\text{v}}\mathcal{L}(f_{\theta_t}(\boldsymbol{x}), y)$, calculate the average validation loss.
 9:         Using equation (5), calculate $r_t$, $b_t$, and $A_t$ from the validation loss $\mathcal{L}_\text{val}$.
10:     end if
11:     if $t$ mod $T_1 = 0$ then
12:         Using equation (6), update the sampler $S_\phi$ with ADAM with the policy loss $\mathcal{L}_\text{policy}$.
13:         Reinitialize the hidden states $\mathbf{h}_\eta, \mathbf{h}_\pi \leftarrow 0$.
14:     end if
15: end for
```

We now explain two submodules of the RLSampler $S = (\eta, \pi)$: a *history logger* η and a *policy network* π. We also provide subtle implementation details in Appendix A.

History Logger The history logger η is a recurrent neural network (RNN) with parameters ϕ_η and hidden states $\mathbf{h}_\eta$. At the previous iteration $t-1$, the module receives an output logit tensor $\boldsymbol{l}_{t-1} := f_{\theta_{t-1}}(\mathcal{B}_{t-1})$ of the batch $\mathcal{B}_{t-1}$ from the learner network f_θ and returns a fixed-length summary $\boldsymbol{s}_t$, which is a state approximation supplied to the policy network at iteration t.

$$(\boldsymbol{s}_t, \mathbf{h}_{\eta_t}) = \eta(\texttt{stopgrad}(f_{\theta_{t-1}}(\mathcal{B}_{t-1})), \mathbf{h}_{\eta_{t-1}}; \phi_{\eta_{t-1}}), \tag{3}$$

where `stopgrad` blocks the backpropagation algorithm to update the values inside, so the update to the sampler does not propagate through the learner. The history logger is implemented with two-layer LSTM cells (Hochreiter & Schmidhuber (1997b)) followed by a single linear decoder.

Policy Network The sampling policy π is a stochastic policy implemented with an RNN. At i-th call to the sampler at iteration t, the policy network receives a summary $\boldsymbol{s}_t$ of the history until the last mini-batch $\mathcal{B}_{t-1}$. The output is a logit vector $\boldsymbol{l}^\pi_{t,i} \in \mathbb{R}^{|\mathcal{X}_\text{t}|}$ corresponding to the probability of selecting each element of the training subset $\mathcal{X}_\text{t}$ for the i-th item of the batch $\mathcal{B}_t$. The probability of sampling an item $\boldsymbol{x}_j \in \mathcal{X}_\text{t}$ follows the categorical distribution corresponds to the logit $\boldsymbol{l}^\pi_{t,i}$, i.e.,

$$\begin{aligned}(\boldsymbol{l}^\pi_{t,i}, \mathbf{h}_{\pi_{t,i}}) &= \pi(\boldsymbol{s}_t, \mathbf{h}_{\pi_{t,i-1}}; \phi_{\pi_{t,i-1}}),\\ p_{\boldsymbol{B}_{t,i}}(j) &= \text{Prob}\{\boldsymbol{B}_{t,i} = \boldsymbol{x}_j | \boldsymbol{s}_t\} = [\texttt{softmax}(\boldsymbol{l}^\pi_{t,i})]_j,\end{aligned} \tag{4}$$

where $\boldsymbol{B}_{t,i}$ is a random variable of selecting an item of the training subset $\mathcal{X}_\text{t}$ for the i-th element of the t-th batch, $p_{\boldsymbol{B}_{t,i}}$ is a probability vector corresponds to it, $\mathbf{h}_{\pi_{t,i}}$ is the hidden state of the recurrent modules of π, and $\phi_{\pi_{t,i}}$ is the network parameter of π. The policy network π consists of a single linear encoder followed by two-layer LSTM cells and a single linear decoder.

3.3 CONTROLLER OBJECTIVE

Our problem setting of finding an optimal optimization framework shares its goal with network architecture search (Zoph & Le (2017); Pham et al. (2018)), from which we borrow the formula of the reward signal. Every reward period T_2, an average validation loss is obtained by evaluating the same loss function $\mathcal{L}$ used to train the learner, but on the entire validation subset $\mathcal{X}_\text{v}$. The reward r_t is defined as an exponential of the averaged validation loss.

$$r_t = \exp\{-2\mathcal{L}_\text{val}\} = \exp\{-2\mathbb{E}_{(\boldsymbol{x},y)\in\mathcal{X}_\text{v}}\mathcal{L}(f_{\theta_t}(\boldsymbol{x}), y)\}, \tag{5}$$

where $\boldsymbol{x}$ is a validation sample, y is its ground truth label, f_{θ_t} is the learner network. We do not backpropagate the gradients inside the reward function.

Using a single reward for the policy gradient update may lead to unnecessarily large variance (Sutton & Barto (2018)). It is common to define a baseline b_t and use the advantage $A_t := r_t - b_t$ to evaluate the policy gradient. Since our policy network is intrinsically sequential, we use exponential moving average of the reward as the baseline (Zoph & Le (2017)), i.e., $b_t = \lambda r_t + (1 - \lambda) b_{t-T_2}$. Here, $\lambda = 0.1$ controls the rate of decay of the received reward.

We use Adam (Kingma & Ba (2015)) to update the sampler $S = (\eta, \pi)$ with the REINFORCE algorithm (Williams (1992)). Because Adam is a stochastic gradient *descent* solver, we define a policy *loss* $\mathcal{L}_{\text{policy}}$, gradient of which is the negative policy gradient. At iteration nT_1, $n \in \mathbb{N}$,

$$\mathcal{L}_{\text{policy},nT_1} = \sum_{m=1}^{T_1/T_2} A_{(n-1)T_1+mT_2} \sum_{l=1}^{T_2} \log \left(\frac{1}{k} \sum_{i=1}^{k} p_{\boldsymbol{B}_{(n-1)T_1+mT_2+l,i}} \right). \tag{6}$$

4 Related Work

Despite the success of deep learning under naïve uniform sampling strategy, several branches of works aim to improve the performance and speed of training. Importance sampling has been studied for convex optimization problems (Bordes et al. (2005); Needell et al. (2014); Zhao & Zhang (2015)), for deep representation learning (Bengio et al. (2009); Alain et al. (2016); Loshchilov & Hutter (2016); Chang et al. (2017); Wu et al. (2017); Katharopoulos & François (2018); Ren et al. (2018); Johnson & Guestrin (2018); Csiba & Richtárik (2018); Peng et al. (2019); Ariafar et al. (2020)), and for deep reinforcement learning (Schaul et al. (2016); Fang et al. (2019)). In this framework, the sampling probability of each training set item is nonuniformly weighted.

Importance sampling is extensively used in reinforcement learning with experience replay (Schaul et al. (2016)), where importance is defined with instantaneous or accumulated rewards. With a stationary dataset, however, the definition of sample importance is rather subtle. Loshchilov & Hutter (2016) use training loss to calculate importance, Wu et al. (2017) define sample weights using a distance metric in the latent space of the learner model, and Katharopoulos & François (2018) utilize the upper bound of the per-sample gradient norm of all the learner parameters to determine the sample importance. Zhao & Zhang (2015) argue that sampling methods that minimize the variance of the stochastic gradients achieve faster convergence for convex optimization problems. However, in high dimensional nonconvex problems such as deep learning, large mini-batch SGD, which *should* have low empirical variance, tends to yield *sharp* minima that generalize worse (Hochreiter & Schmidhuber (1997a); Keskar et al. (2017)). Instead of using heuristics to define sample weights, we allow the sampler to be a trainable agent that adaptively learns the sampling distribution.

There are other branches of works on sampling strategy. Our method shares its philosophy with active sampling (Bengio & Senecal (2008); Li & Guo (2013)), where a learning agent chooses which data sample to learn next; however, in our problem setting, we do not presume an external annotator. Determinantal point process (DPP, Dereziński et al. (2019); Huang et al. (2019); Zhang et al. (2019)) is a well-used technique in machine learning. In DPP, the sampling process is viewed as a point process on the sample space. Despite their firm mathematical background, they use additional optimization procedure to preprocess the dataset and construct a kernel matrix with size $N \times N$, where $N = |\mathcal{X}|$, which becomes intractable in modern deep learning frameworks.

Another branch of methods that modifies training schedule in order to improve the quality of the learning is to provide a sequence of *subsets* of a training set. Curriculum learning (CL, Bengio et al. (2009); Graves et al. (2017); Hacohen & Weinshall (2019)) methods start by exposing the learner to small and easier subsets of the training set and gradually increase its size, to make the learner experiences harder examples in the later steps. Self-paced learning (SPL, Kumar et al. (2010); Shrivastava et al. (2016)), on the other hand, stresses on harder samples to train the network. Among those, Fan et al. (2017) is the closest to our method. They use an RL agent to filter out samples from a mini-batch for CL; however, we use an RL agent to directly sample from the training set.

5 Evaluation

We begin by testing our RLSampler over common image classification benchmarks: CIFAR-10 and CIFAR-100 (Krizhevsky (2009)). We then demonstrate the effectiveness of RLSampler in more constrained environments, i.e., in case with larger learning rates to quantify the training stability the sampler can provide. Generated sequences of indices from the evaluations are further analyzed in

Table 1: Top-1 classification error on CIFAR-10.

Model	Random Sampler			**RLSampler (Ours)**			Accuracy Gain
	Mean±SD	Best	Worst	Mean±SD	Best	Worst	
VGGNet-16	9.082±0.119	8.94	9.26	8.824±0.096	8.74	8.99	**0.258** (2.84%)
VGGNet-16 + BN	8.016±0.140	7.81	8.25	7.772±0.203	7.46	8.04	**0.244** (3.04%)
ResNet-20	8.658±0.246	8.33	9.02	7.804±0.249	7.52	8.22	**0.854** (9.86%)
WRN-28-2	5.486±0.141	5.22	5.62	5.204±0.131	4.97	5.34	**0.282** (5.14%)
WRN-28-10	4.056±0.102	3.93	4.24	3.704±0.090	3.58	3.82	**0.352** (8.68%)

Table 2: Top-1 classification error on CIFAR-100.

Model	Random Sampler			**RLSampler (Ours)**			Accuracy Gain
	Mean±SD	Best	Worst	Mean±SD	Best	Worst	
VGGNet-16	34.520±0.370	33.88	34.88	33.750±0.621	32.94	34.82	**0.770** (2.23%)
VGGNet-16 + BN	31.340±0.292	31.06	31.85	30.478±0.271	30.06	30.81	**0.862** (2.75%)
ResNet-20	33.846±0.511	32.99	34.58	32.606±0.421	31.87	33.17	**1.240** (3.66%)
WRN-28-2	26.700±0.207	26.40	26.98	25.540±0.326	25.18	25.98	**1.160** (4.34%)
WRN-28-10	20.040±0.109	19.89	20.19	18.918±0.195	18.65	19.24	**1.122** (5.60%)

Section 6. All experiments are done with PyTorch (Paszke et al. (2019)). The code will be available after the publication.

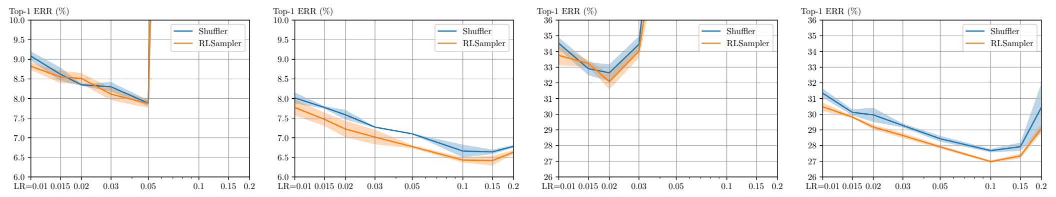
5.1 Standard Image Classification

Both CIFAR-10 and CIFAR-100 (Krizhevsky (2009)) are widely used benchmarks, each contains 50k training and 10k test images, equally distributed in 10 and 100 classes, respectively. To verify if our RLSampler can generalize well with various model architectures, we train two 16-layer VGGNets (Simonyan & Zisserman (2015)) with and without batch normalization layers (Ioffe & Szegedy (2015)), a 20-layer ResNet (He et al. (2016)), a WRN-28-2, and a WRN-28-10 (Zagoruyko & Nikos (2016)). For each experiment, we train five times with the same hyperparameters but with different initializations of the learner and the sampler. We report the mean, the standard deviation, the best and the worst top-1 error in Tables 1 and 2. In the cases with an RLSampler, we spare the last 640 (1.28%) samples of 50k training set for the validation subset of CIFAR-10, and the last 1280 (2.56%) samples for the CIFAR-100. For a random sampler without replacement, we use the original 50k training set. More details of the architectures and other hyperparameters used in the experiments are summarized in Appendix B.

Results in Tables 1 and 2 show that networks trained with our RLSampler consistently outperform the ones trained with a random sampler without replacement. Although the learner with an RLSampler experiences fewer training examples due to additional train/validation split, in some cases, like the case of WRN-28-10 on both datasets, even the worst case with an RLSampler shows better performance than the best case with a random sampler. With RLSampler we achieved a maximum average gain of 9.9% on CIFAR-10 (ResNet-20) and 5.6% on CIFAR-100 (WRN-28-10).

5.2 Training with Increasing Learning Rate

The problem of exploding and vanishing gradients in a deep network has been a major hurdle to achieve stable training. This problem is largely alleviated by the introduction of residual connections (He et al. (2016)) and batch normalization (BN, Ioffe & Szegedy (2015)). Recent works (Bjorck et al. (2018); Santurkar et al. (2018)) explain that the effectiveness of BN comes from maintaining good stability under a higher learning rate. We conduct a series of evaluations to see whether an RLSampler delivers a similar effect. Two 16-layer VGGNets with and without BN are trained on CIFAR-10 with various learning rates. Figure 2 shows average top-1 errors and standard deviations of each setting, based on three evaluations with different model initializations. It turns out that the advantage of using RLSampler is subtle in the network without BN; the benefits of BN and adequate sampling strategy are additive.



(a) VGG-16 on CIFAR-10. (b) VGG-16-BN on CIFAR-10. (c) VGG-16 on CIFAR-100. (d) VGG-16-BN on CIFAR-100.

Figure 2: Image classification results of VGGNets (Simonyan & Zisserman (2015)) with two different samplers and various learning rates (LRs). We report the average (bold center line) and the standard deviation (shaded area) of three runs, except for the base LR (0.01) case, where we use the values in Tables 1 and 2.

6 Analysis on the Sampled Sequence

In this section, we apply pre-trained RLSampler models and pre-sampled sequences of training mini-batches generated in the main evaluations in Section 5.1 to different learner initializations and architectures. The objective of this analysis is to demonstrate the versatility of the generated sample sequence. We conclude our analysis by inspecting the internal statistics of the sampled sequence.

6.1 Model-Independent Optimal Sample Sequence

We conduct two additional experiments to show that the effectiveness of the generated training sample sequence is more an intrinsic property of the dataset and is independent to the models being trained. We use pre-trained RLSampler models and pre-sampled sequences from the best one among five runs shown in Tables 1 and 2 in the subsequent analyses.

Cross-Initialization Experiments Using an RLSampler and a training sequence used to train the best ResNet-20 model on CIFAR-10 in Section 5.1, we train 10 different initializations of the identical learner architecture. The result from each network instance is compared to the top-1 error that the same initialization obtains under a random sampler, averaged over 10 runs. We compare three settings: the two with the pre-trained RLSampler, having its weights fixed and not fixed, and the one with the exact same sequence of indices used to train the best model. Table 3 summarizes the result. Although the sampler and the sequence are derived from a different training framework, we obtain a consistent performance boost compared to a naïve random sampler.

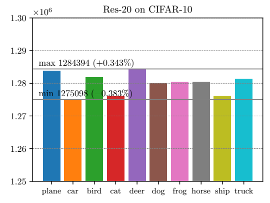
Table 3: Cross-initialization experiment results.

Imported	Before	After	**Avg. Gain±SD**
Fixed RLSampler	8.723	8.174	0.549±0.362
Unfixed RLSampler	8.697	8.096	0.601±0.282
Samples Only	8.631	8.064	0.567±0.241

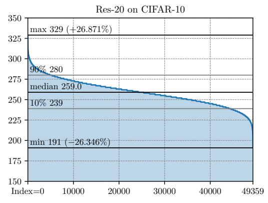
Cross-Architecture Experiments We train various convolutional neural networks on CIFAR-10 with the pre-generated sample sequences used to train the best models in Table 1. The results are summarized in Table 4, where we report the average top-1 error over five runs. the gain is calculated as a difference between this results and the average top-1 error of naïve random sampling in Table 1. The first column in Table 4 is the model the pre-generated sample sequence is obtained from, and the second column is the different architecture that the same sequence is applied to for training. Even though the architectures are different from the models for which the training sample sequences are optimized, again, the pre-sampled sequence provides consistent performance gain, except for the case when VGGNet without BN is a recipient of the training sequence. However, the optimized sequence for VGGNet-16 without BN is still effective in training other networks with BN, e.g., ResNet and WRN.

Table 4: Cross-architecture experiment results.

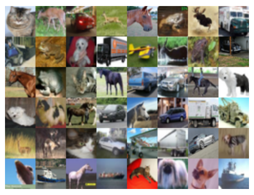
Sequence from	→ Applied to	Top-1 Error±SD	**Gain**
ResNet-20	→ VGG-16	9.094±0.170	-0.012
WRN-28-10	→ VGG-16	9.080±0.040	0.002
ResNet-20	→ VGG-16-BN	7.826±0.043	0.190
WRN-28-10	→ VGG-16-BN	7.622±0.031	0.394
VGG-16	→ ResNet-20	8.204±0.114	0.454
VGG-16-BN	→ ResNet-20	8.104±0.201	0.554
WRN-28-10	→ ResNet-20	8.102±0.113	0.556
VGG-16-BN	→ WRN-28-10	3.733±0.074	0.323
ResNet-20	→ WRN-28-10	3.656±0.078	0.400



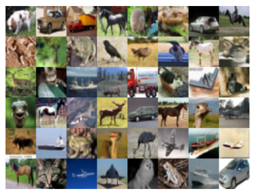
(a) Class frequency.



(b) Sample frequency.



(c) 48 most sampled images



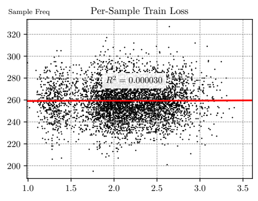
(d) 48 least sampled images

Figure 3: The sample distribution induced by our RLSampler. (a) and (b) show class and sample frequency of the entire training sequence of the best ResNet-20 model on CIFAR-10 in Tabel 1, respectively. For the sample frequency in (b), with the maximum and the minimum, we also provide the 10-th, 50-th (median), and 90-th percentiles. We report the percentage of deviation of the two extrema from the ideal mean under the uniform distribution. (c) and (d) show the most and the least fetched images from the same RLSampler, respectively.

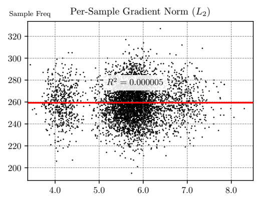
6.2 Internal Structure of the Sampled Sequence

Emerged Nonuniform Sample Weights Figures 3a and 3b visualize the class and the sample distributions generated by RLSampler. RLSampler samples each class with a relatively small variance; in CIFAR-10, the frequency difference between the most and the least sampled classes are less than 1%. However, RLSampler develops a clear preference over particular samples, even though we do not introduce any prior knowledge of the data distribution. As shown in Figure 3b, the top 10% of the samples are fetched at least 15% more frequently than the bottom 10%.

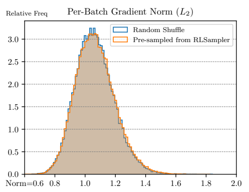
Agreements with Previous Intuitions We are interested in the relationship between the distributional properties that emerge by training RLSampler and the heuristics in Section 4 for designing intelligent sampling methods. We use 10 differently initialized ResNet-20 trained for only 5 epochs as the baseline to evaluate the intermediate statistics in Figure 4. We plot the relationship between the sample frequency and per-sample training loss (Loshchilov & Hutter (2016)), as well as per-sample gradient norm (Katharopoulos & François (2018)) in Figures 4a and 4b, respectively. As results of a simple linear regression show that learned sample weights of our RLSampler do not correlate with these factors with a per-sample basis. Moreover, as in Figure 4c, statistics of intermediate gradient norms of the learner with respect to the generated mini-batches by the RLSampler is indistinguishable from the ones from a naïve random sampling. Although this does *not* falsifies the intuitions of the previous methods (Loshchilov & Hutter (2016); Katharopoulos & François (2018)), the results strongly implies the existence of other factors determining the goodness of a sample sequence.



(a) vs. training loss.



(b) vs. gradient norm.



(c) Histogram of gradient norms.

Figure 4: (a) and (b): Empirical sampling probability of RLSampler compared to the average training loss and the average L_2 gradient norm of ten different ResNet-20 fixed after being trained for 5 epochs, respectively. (c): Histogram of the L_2 gradient norms with respect to sample mini-batches from two different samplers, calculated on the same networks as in (a) and (b). The batch size is the same (128) as the experiments in Table 1. The data for the histogram in (c) is obtained by collecting the gradient norms of the fixed network fed by the first 4k mini-batches fetched from each pre-generated sequence.

7 Conclusion

We investigated using a trainable adaptive dataset sampling module for a deep learning framework. We constructed an effective model-agnostic adaptive sampler, dubbed RLSampler, based on a deep policy gradient algorithm. RLSampler shows clear improvements of generalization errors on image classification benchmarks. Additional experiments on the trained RLSampler strongly imply the existence of a model-agnostic optimal sampling sequence of a given dataset for deep learning.

REFERENCES

Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. YouTube-8M: A large-scale video classification benchmark. *arXiv preprints*, September 2016.

Guillaume Alain, Alex Lamb, Chinnadhurai Sankar, Aaron Courville, and Yoshua Bengio. Variance reduction in SGD by distributed importance sampling. In *ICLR Workshop*, 2016.

Setareh Ariafar, Zelda Mariet, Ehsan Elhamifar, Dana Brooks, Jennifer Dy, and Jasper Snoek. Faster & more reliable tuning of neural networks: Bayesian optimization with importance sampling. 2020. Technical Report.

Y. Bengio and J. Senecal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722, 2008.

Yoshua Bengio, Jérôme Louradour, and Ronan Collobert. Curriculum learning. In *ICML*, 2009.

Johan Bjorck, Carla Gomes, Bart Selman, and Kilian Q. Weinberger. Understanding batch normalization. In *NIPS*, 2018.

Antoine Bordes, Seyda Ertekin, Jason Weston, and Léon Bottou. Fast kernel classifiers with online and active learning. *Journal of Machine Learning Research*, 6:1579 – 1619, September 2005.

Haw-Shiuan Chang, Erik Learned-Miller, and Andrew McCallum. Active bias: Training more accurate neural networks by emphasizing high variance samples. In *NIPS*, 2017.

Dominik Csiba and Peter Richtárik. Importance sampling for minibatches. *Journal of Machine Learning Research*, 19(1):1 – 21, August 2018.

Michał Dereziński, Daniele Calandriello, and Michal Valko. Exact sampling of determinantal point processes with sublinear time preprocessing. In *NIPS*, 2019.

Yang Fan, Fei Tian, Tao Qin, Jiang Bian, and Tie-Yan Liu. Learning what data to learn. In *ICLR Workshop*, 2017.

Meng Fang, Tianyi Zhou, Yali Du, Lei Han, and Zhengyou Zhang. Curriculum-guided hindsight experience replay. In *NIPS*, 2019.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

Alex Graves, Marc G. Bellemare, Jacob Menick, Rémi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *ICML*, 2017.

Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. In *ICML*, 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural Computation*, 9(1):1 – 42, 1997a.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8): 1735 – 1780, 1997b.

Wanming Huang, Richard Yi Da Xu, and Ian Oppermann. Efficient diversified mini-batch selection using variable high-layer features. In *ACML*, 2019.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

Tyler B. Johnson and Carlos Guestrin. Training deep models faster with robust, approximate importance sampling. In *NIPS*, 2018.

Angelos Katharopoulos and Fleuret François. Not all samples are created equal: Deep learning with importance sampling. In *ICML*, 2018.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. In *ICLR*, 2017.

Diederik P. Kingma and Jimmy Lei Ba. Adam: a method for stochastic optimization. In *ICLR*, 2015.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009. Technical Report.

M. Pawan Kumar, Benjamin Packer, and Daphne Koller. Self-paced learning for latent variable models. In *NIPS*, 2010.

X. Li and Y. Guo. Adaptive active learning for image classification. In *CVPR*, 2013.

Ilya Loshchilov and Frank Hutter. Online batch selection for faster training of neural networks. In *ICLR Workshop*, 2016.

Deanna Needell, Rachel Ward, and Nati Srebro. Stochastic gradient descent, weighted sampling, and the randomized Kaczmarz algorithm. In *NIPS*, 2014.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: an imperative style, high-performance deep learning library. In *NIPS*, 2019.

Xinyu Peng, Li Li, and Fei-Yue Wang. Accelerating minibatch stochastic gradient descent using typicality sampling. *IEEE Transactions on Neural Networks and Learning Systems (Early Access)*, pp. 1 – 11, December 2019.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff . Efficient neural architecture search via parameter sharing. In *ICML*, 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, February 2019. Technical Report.

Mengye Ren, Wenyuan Zeng, Bin Yang, and Raquel Urtasun. Learning to reweight examples for robust deep learning. In *ICML*, 2018.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. *IJCV*, 115(3):211–252, 2015.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? In *NIPS*, 2018.

Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *ICLR*, 2016.

Abhinav Shrivastava, Abhinav Gupta, and Ross Girshick. Training region-based object detectors with online hard example mining. In *CVPR*, 2016.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2 edition, November 2018. ISBN 9780262039246.

Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229 – 256, 1992.

D. Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429 – 1451, 2003.

Chao-Yuan Wu, R. Manmatha, Alexander J. Smola, and Philipp Krähenbühl. Sampling matters in deep embedding learning. In *ICCV*, 2017.

Sergey Zagoruyko and Komodakis Nikos. Wide residual networks. In *BMVC*, 2016.

Cheng Zhang, Cengiz Öztireli, Stephan Mandt, and Giampiero Salvi. Active mini-batch sampling using repulsive point processes. In *AAAI*, 2019.

Peilin Zhao and Tong Zhang. Stochastic optimization with importance sampling for regularized loss minimization. In *ICML*, 2015.

Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. In *ICML*, 2017.

A Implementation Details

In addition to Section 3, we further describe the implementation details of RLSampler models. First, the detailed behavior of each recurrent module of the RLSampler is provided. Next, we provide the hyperparameters used in the experiments in Sections 5 and 6.

A.1 Controller Architecture

As described in Section 3.2, our RLSampler S_ϕ consists of two LSTM (Hochreiter & Schmidhuber (1997b)) networks: a history logger and a policy network. The history logger consists of two LSTM cells, followed by a single linear decoder. Each LSTM cell $\texttt{lstm}_i, i \in \{1, 2\}$ has a hidden state vector $\mathbf{h}^{(i)}$ and a cell state vector $\mathbf{c}^{(i)}$ which are fed into the LSTM cell along with the external input $\mathbf{x}^{(i)}$, i.e., at iteration $t-1$,

$$(\mathbf{h}_t^{(i)}, \mathbf{c}_t^{(i)}) = \texttt{lstm}_i(\mathbf{x}_{t-1}^{(i)}, \mathbf{h}_{t-1}^{(i)}, \mathbf{c}_{t-1}^{(i)}). \tag{7}$$

At the beginning and every policy update, these state vectors $\mathbf{h}^{(i)}, \mathbf{c}^{(i)}$ of each LSTM cell of the history logger is reset to zero to limit the amount of historical gradients to store. At iteration $t-1$, logits $\boldsymbol{l}_{t-1} = (\boldsymbol{l}_{t-1,1}, \boldsymbol{l}_{t-1,2}, \cdots, \boldsymbol{l}_{t-1,k})$ from the learner network is averaged batch-wise and fed into the input of the first LSTM cell $\texttt{lstm}_1$. The second LSTM cell $\texttt{lstm}_2$ receives the hidden state $\mathbf{h}_1$ of the first cell $\texttt{lstm}_1$ as an input. The hidden state $\mathbf{h}_2$ of the second LSTM cell $\texttt{lstm}_2$ is then fed into the decoder $\texttt{dec}$ to produce the current historical summary. Therefore, the internal logic of the equation (3), or a single-step update of the history logger, is:

$$\begin{aligned}
(\mathbf{h}_t^{(1)}, \mathbf{c}_t^{(1)}) &= \texttt{lstm}_1\left(\frac{1}{k}\sum_{j\in\mathcal{B}_{t-1}} \boldsymbol{l}_j, \mathbf{h}_{t-1}^{(1)}, \mathbf{c}_{t-1}^{(1)}\right), \\
(\mathbf{h}_t^{(2)}, \mathbf{c}_t^{(2)}) &= \texttt{lstm}_2\left(\mathbf{h}_t^{(1)}, \mathbf{h}_{t-1}^{(2)}, \mathbf{c}_{t-1}^{(2)}\right), \\
\boldsymbol{s}_t &= \texttt{dec}\left(\mathbf{x}_{t-1}^{(i)}, \mathbf{h}_{t-1}^{(i)}, \mathbf{c}_{t-1}^{(i)}\right),
\end{aligned} \tag{8}$$

where $\boldsymbol{s}_t$ is the summary of the history of the learner network states available at iteration t, and $k = |\mathcal{B}|$ is a batch size. We initialize each bias term of the LSTM cells and the linear layer to be zero. Input-to-hidden weights of the LSTM cells are initialized with Xavier initialization (Glorot & Bengio (2010)), and hidden-to-hidden weights of the LSTM cells are initialized to a random orthogonal matrix. The linear layer weight is initialized with zero-mean normal distribution with a standard deviation of 1e-3.

The policy network has an identical architecture as the aforementioned history logger, except it has a linear layer (an encoder) at the input. Each layer and the hidden states of the policy network are initialized exactly the same as the history logger except for the linear layer weights, which are initialized with zero-mean normal distribution with standard deviation of 1e-5. The policy network receives only the historical summary and outputs a logit vector corresponding to the sampling probability of each item in the training subset. We use the Adam optimizer (Kingma & Ba (2015)) to train both submodules $S_\phi = (\eta, \pi)$, with the default parameters $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon =$ 1e-8, and zero weight decay.

A.2 Hyperparameters

The hyperparameters are chosen by manual searching based on the performance of a 20-layer ResNet (He et al. (2016)) on CIFAR-10 (Krizhevsky (2009)) dataset. For the hyperparameter search, we split the 50k training set into training/validation set containing 45000/5000 items. After the hyperparameters are chosen, they are fixed for all the experiments and analyses in Sections 5 and 6. The only difference in hyperparameters is the size of the validation subset $|\mathcal{X}_\text{v}|$, which is affected by the number of classes the dataset $\mathcal{X}$ has. The effectiveness of the chosen architecture and the hyperparameters across various learner models proves that RLSampler can be viewed as a model-agnostic extension to the existing deep learning frameworks. The chosen hyperparameters are presented in Table 5.

Table 5: Hyperparameters of RLSampler used for evaluation.

Parameter name	Parameter value
Validation subset size, $\|\mathcal{X}_\text{v}\|$	640 (CIFAR-10) or 1280 (CIFAR-100)
Reward period, T_2	20
Policy update period, T_1	100
Summary dimension, dim$(\boldsymbol{s})$	64
Hidden state dimension of the history logger, dim$(\mathbf{h}_\eta)$	256
Hidden state dimension of the policy network, dim$(\mathbf{h}_\pi)$	64
Reward decay rate, λ	0.1
Sampler optimizer	ADAM$(\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = $ 1e-8)
Sampler learning rate	0.04
Sampler scheduling	Uniform

B Experimental Settings

In this section, details of the experimental settings of Sections 5 and 6 are provided. For the evaluation in Section 5.1, we choose convolutional neural network models that have achieved previous state-of-the-art in major challenge, especially the ImageNet Large Scale Visual Recognition Challenge (ILSVRC, Russakovsky et al. (2015)). The models used in the evaluations are summarized in Table 6. To show the versatility of our RLSampler, we use an identical sampler architecture across all experiments. We also match each component of a learning framework, i.e., a loss function including weight decay, an image preprocessing procedure, an optimization algorithm, a learning rate schedule, the size of a mini-batch, and the total number of iterations, to those mentioned in the original papers (Simonyan & Zisserman (2015); He et al. (2016); Zagoruyko & Nikos (2016)). For training, we prepare both datasets by normalizing with the data mean and standard deviation, applying random horizontal flips and random crop with 4-pixel pad on every side. Especially, we use reflective padding to augment images for WRN-28-2 and WRN-28-10; otherwise, we use zero padding. Note that this augmentation is applied to both splits $\mathcal{X} = (\mathcal{X}_\text{t}, \mathcal{X}_\text{v})$ of the training set. Behaviors of dropout (Srivastava et al. (2014)) and batch normalization (Ioffe & Szegedy (2015)) layers are also the same in both splits $(\mathcal{X}_\text{t}, \mathcal{X}_\text{v})$. For the test set $\mathcal{X}_\text{test}$, we only do the normalization.

However, there are few differences. For ResNets (He et al. (2016)) we trained longer than the original version, with a modified learning rate schedule. In addition, Simonyan & Zisserman (2015) do not provide evaluations of VGGNets on CIFAR benchmarks; therefore, we use the model designed for ImageNet (Russakovsky et al. (2015)) with fully connected layers modified to have lower dimensions. We reduce the dimension of three fully connected layers of a VGGNet to be 512. The overall optimization settings are summarized in Table 7.

For RLSampler, a train/validation split of 49360/640 for CIFAR-10 and 48720/1280 for CIFAR-100 is used. This means that the sampler is trained with 5 validation batches for CIFAR-10 and 10 batches for CIFAR-100. We simply use the samples with the latest indices in the original training set $\mathcal{X}$ as the validation samples. For fair comparison, we match the number of learner updates t for the time horizon T_0 as well as the learning rate schedule between the same models trained with different samplers. Every instance of experiments is done on a single NVIDIA RTX 2080 Ti GPU.

Table 6: Learner models used for evaluation and their key features.

Model name	# Layers	BatchNorm	Residual Connection	Dropout	Reference
VGGNet-16	16	✗	✗	0.5	Simonyan & Zisserman (2015)
VGGNet-16 + BN	16	✓	✗	0.5	Ioffe & Szegedy (2015)
ResNet-20	20	✓	✓	✗	He et al. (2016)
WRN-28-2	28	✓	✓	✗	Zagoruyko & Nikos (2016)
WRN-28-10	28	✓	✓	✗	Zagoruyko & Nikos (2016)

Table 7: Detailed experimental settings for each of the main evaluations. We use the exact same settings for the experiments on CIFAR-10 and on CIFAR-100, except for the size of the validation subset $\mathcal{X}_\text{v}$.

	VGGNet-16	VGGNet-16 + BN	ResNet-20	WRN-28-2	WRN-28-10
# training steps, T_0	80k	80k	100k	50k	78k
Base LR	0.01	0.01	0.1	0.1	0.1
LR schedule	[40k, 60k]	[40k, 60k]	[50k, 75k]	[20k, 40k]	[23k, 47k, 63k]
LR decay coefficient	0.1	0.1	0.1	0.2	0.2
Nesterov momentum	✗	✗	✗	✓	✓
Momentum	0.9	0.9	0.9	0.9	0.9
Weight decay	1e-4	1e-4	1e-4	5e-4	5e-4