In-Loop Meta-Learning with Gradient-Alignment Reward

Samuel Müller,[1,*] **André Biedenkapp,**[1] **Frank Hutter,**[1, 2]
[1]Department of Computer Science, University of Freiburg, Germany
[2]Bosch Center for Artificial Intelligence, Renningen, Germany
{muellesa, biedenka, fh}@cs.uni-freiburg.de

Abstract

At the heart of the standard deep learning training loop is a greedy gradient step minimizing a given loss. We propose to add a second step to maximize training generalization. To do this, we optimize the loss of the next training step. While computing the gradient for this generally is very expensive and many interesting applications consider non-differentiable parameters (e.g. due to hard samples), we present a cheap-to-compute and memory-saving reward, the *gradient-alignment reward (GAR)*, that can guide the optimization. We use this reward to optimize multiple distributions during model training. First, we present the application of GAR to choosing the data distribution as a mixture of multiple dataset splits in a small scale setting. Second, we show that it can successfully guide learning augmentation strategies competitive with state-of-the-art augmentation strategies on CIFAR-10 and CIFAR-100.

1 Introduction

The human capacity to learn is staggering. Not only can humans learn about the world they live in, but crucially humans can also learn a a good learning strategy simultaneously. To this end, humans are able to learn to select their learning path.

Take a math course as an example. In the beginning you might have followed your professor's textbook closely. With increasing understanding, you might have realized there are other textbooks out there, which yield better learning outcomes. In the same course you might also have learned that it can be helpful to try to prove the presented theorems yourself, instead of only reading the proof provided in the textbook. You might have learned both of these learning strategies while learning the material of the course at the same time. So, your understanding of the material and the way you studied both improved as you studied. Similarly, a promising route towards stronger machine learning models could lie in training not only for performing a particular task, but for better learning strategies on that task at the same time.

We define an improved learning method as a method that yields outcomes that generalize well to unseen problem instances. In a supervised learning setting that means validation error is minimized. One way to achieve this is by optimizing meta-parameters with a meta-gradient. The meta-gradient is a gradient taken through an SGD training run, which is possible since SGD itself is differentiable.

We define the meta-loss for the current step in a training loop as the loss in the next step, after performing an SGD step. Thus, the meta-loss, unlike the loss, decreases if a step on some data leads to improved performance on different data. This meta-loss can be particularly useful for parts of the training algorithm that do not make sense to be directly optimized with the usual loss. A particular example for this is the data distribution. While it was shown that it can be helpful to change the data distribution (over time) for SGD training, by training with a curriculum (see e.g. Bengio, LeCun et al. 2007) or by filtering the dataset (see e.g. Chen et al. 2019), we can generally not decide what data to train on using the same loss we use for model training: this would simply encourage the model to focus on a particularly easy data distribution, since that would minimize the loss. The meta-loss on the other hand rewards generalization. The example of learning a distribution over a dataset is also particularly interesting because it is not differentiable in general. Thus, we would not be able to compute meta-gradients, even if compute cost was no issue.

We evaluate our method for two settings: learning a distribution over data splits in a toy experiment and augmentation selection in a real-world application to CIFAR-10 and CIFAR-100 (Krizhevsky, Hinton et al. 2009).

The over-arching contribution of this paper is to introduce a way of adapting meta parameters ϕ online during SGD training of the elementary network parameters θ that is efficient in terms of data, memory and time. We present a method to train online for generalization and do so for non-differentiable distributions. Specifically, our contributions are:

- We present a mechanism to cheaply approximate meta-gradients in SGD, using the next batch as a proxy for the validation set.
- We introduce the Gradient-Alignment Reward setup with an efficient implementation, which allows us to use reinforcement learning with an unique per-example reward.
- We show promising empirical results for our approach on a simple example and a real-world application.

[*]Contact Author

Algorithm 1 In-Loop Meta-Learning

```
1: initialize parameters $\theta_1, \phi_1, \phi_2 := \phi_1$
2: for $t \in \{1, \ldots, T\}$ do
3:     $l_t \leftarrow \ell(\theta_t, \phi_t)$
4:     $\theta_{t+1} \leftarrow \theta_t - \alpha \nabla_{\theta_t} l_t$
5:     if $t > 1$ then
6:         $\phi_{t+1} \leftarrow \mathrm{sg}(\phi_t - \beta \nabla_{\phi_{t-1}} l_t)$
7:     end if
8: end for
```

2 Related Work

A commonly-used approach in the literature (e.g., Luketina et al. (2016) or Liu, Simonyan, and Yang (2019)) is to use *alternating SGD*, which alternates SGD steps for θ w.r.t. training loss and for ϕ w.r.t. validation loss. We follow a similar approach with two key differences: (i) we do not use a held-out dataset to train ϕ and (ii) we train weights ϕ parameterizing a distribution non-differentiable p.

Most closely related to our proposed method is the work by Wang et al. (2020) on data weighting. In this work they propose a similar reward to ours here, with two key differences: (i) they only consider aligning with the gradients of validation examples and (ii) they only use an approximation to the example-wise alignment. Counter, instead of using an approximation, we present a very efficient method of computing the example-wise alignments exactly (see Section 5). Further, we discuss properties of the reward in detail (see Section 4), which prior has been omitted.

Other related work stems from the realm of online curriculum learning (see e.g. Graves et al. 2017). Similarly to our toy example, this line of work decides on what data to train as the training goes. The main difference is that we directly optimize the data distribution using the aforementioned approximation of the meta-gradient.

3 In-Loop Meta-Training

In contrast to previous work, in order to be more data-efficient and not require a validation set, we propose to exploit the fact that we are using SGD, and that we can use the next batch as a cheap proxy for the validation set.

Algorithm 1 outlines the general approach. The standard SGD loop is shown in black. In each step the parameters θ are optimized to greedily minimize a stochastic loss $\ell(\theta)$. In red we extend this standard framework with meta-learning updates. The loss now depends additionally on the meta parameters ϕ. We optimize ϕ not to minimize the loss directly, but to minimize the loss of the next step through the update performed in this step, as is done in standard unrolled gradient loop setups. In this setup we can have a direct dependence of the next-step loss on the meta parameters ϕ not through the update, since the module that ϕ parameterizes might be applied in that step, too (this is, e.g., the case if ϕ parameterizes the data augmentation strategy, a case we tackle in our experiments). Since we propose to re-use the training steps as validation steps, we have a new dependency of ϕ compared to previous work, namely on the validation loss directly. This results, for differentiable setups, in the following meta gradient:

$$\nabla_{\phi_{t-1}} l_t = \nabla_{\theta_t} l_t \cdot \nabla_{\phi_{t-1}} \theta_t + \nabla_{\phi_t} l_t \cdot \nabla_{\phi_{t-1}} \phi_t.$$

The first term here describes the meta-gradient we are interested in: how to change ϕ_{t-1} such that the next update of θ improves the loss in the following step t. The second term on the other hand describes how to change ϕ_{t-1} such that its update ϕ_t makes step t as simple as possible; this does not facilitate generalization. We therefore propose to either just use a different loss in every second step, which does not depend on ϕ (e.g., in our case of ϕ parameterizing the data augmentation, just use a simple default augmentation in every second step), or cancel the second term artificially. In Algorithm 1 we show the second option: to cancel the term, we detach ϕ from the graph after each update, as is indicated in the algorithm by the stop gradient operation sg. The sg function is defined as the identity, but with a zero gradient; so $sg(x) = x$ for all x, but $\frac{\partial sg(x)}{\partial x} = \mathbf{0}$.

While this setup is very general, it also is very expensive to compute the meta-gradient $\nabla_{\phi_{t-1}} l_t$. As validation for this statement we performed a small experiment with a WideResNet-28-10 on an NVIDIA Tesla P100. We average step times over one epoch of CIFAR10 training. We looked at two ways of computing the meta-gradients. (i) First, we used the `higher` library (Grefenstette et al. 2019) for the meta-gradient computations. (ii) Second, we re-used the gradient $\nabla_{\theta_t} \ell_t$ to compute $\nabla_{\phi_{t-1}} \ell_t$ using the chain rule through an SGD update $\nabla_{\phi_{t-1}} \ell_t = -\alpha \nabla_{\theta_t} \ell_t \cdot \nabla_{\phi_{t-1}} \ell_{t-1}$. The optimized version improved the memory footprint slightly over the `higher` implementation, yielding a $2.7\times$ memory-increase compared to a $3.1\times$ memory-increase, but both had a very comparable step time-increase of around $6.4\times$. Both the time and the memory overhead are a problem for large-scale machine learning training runs. A training run that finishes over the weekend without meta-gradients might take over half a month with this direct implementation of online meta-learning. Additionally, potential memory problems might require changes to the training pipeline as it would require nearly triple the memory. Most importantly, back-propagation can only compute the derivative to fully differentiable parts of the training process. In this work we follow the setup of Algorithm 1, but propose a reward to approximate this gradient for any distribution.

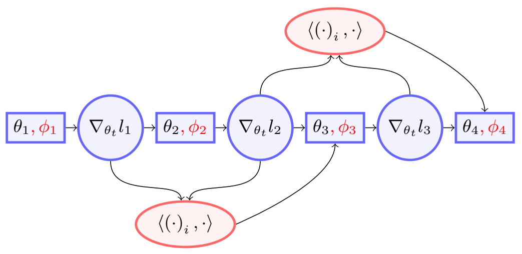


Figure 1: A diagram outlining in-loop meta-learning with the GAR. Extensions to the standard SGD loop are red.

4 Gradient-Alignment Reward

As discussed above there are cases where it is not possible to compute the meta-gradient directly, because we optimize some distribution p that depends on the meta-parameters ϕ and produces hard samples $a_i \sim p(\cdot;\phi)$ for each data point. For example, a_i could represent a network hyperparameter or the data sampling strategy. In these cases, a simple approximation to the meta-gradient is to use the REINFORCE trick (Williams 1992) with the negation of the next step's loss as reward $\overline{r_t} = -\mathcal{L}(\theta_{t+1})$ using

$$\nabla_{\phi_t} \mathop{\mathbb{E}}_{a\sim p(\cdot;\phi_t)}[\mathcal{L}(\theta_{t+1})] \approx \overline{r_t} \cdot \sum_{i=1}^{n} \nabla_{\phi_t} \log p(a_i;\phi_t),$$

where $\mathcal{L}(\theta_t)$ is the batch loss of the t-th step and $a \sim p(\cdot;\phi_t)$. We refer to this approximation of the meta-gradient as *Next Step Loss Reward (NSLR)*. While this approximation is bias-free and simple, it incurs a lot of variance. Further, it provides a single reward only, even though we sample from p for each data point. It would be far more effective to use a reward for each sample instead of one reward for the whole batch. To achieve efficient online meta-learning, we propose the *Gradient-Alignment Reward (GAR)* which allows us to use reinforcement learning with a unique per-example reward.

GAR allows to do efficient in-loop meta-learning for large models and large datasets. It is computed mostly from artifacts of a standard SGD training and has little memory overhead. We define the GAR as the dot product of a current example-gradient with the next step's gradient. Formally, the GAR $r_{t,i}$ for the i-th example in step t is

$$r_{t,i} = \langle \nabla_\theta \ell(\theta_t, \phi_t)_i, \nabla_\theta \mathcal{L}(\theta_{t+1}) \rangle, \tag{1}$$

where $\ell(\cdot,\cdot)_i$ is the loss of the i-th example in a batch, n is the batch size and $\mathcal{L}(\theta_{t+1}) = \frac{1}{n}\sum_{j=1}^{n} \ell(\theta_{t+1}, \phi_{t+1})_j$ is the batch loss of the $(t+1)$-th step. We maximize this reward by sampling from a policy parameterized by ϕ for each example and compute a gradient estimate with the REINFORCE trick. Using vanilla policy-gradient the gradient toward ϕ_t is estimated as

$$\nabla_{\phi_t} \mathop{\mathbb{E}}_{a\sim p(\cdot;\phi_t)}[\mathcal{L}(\theta_t - \alpha\nabla_{\theta_t}\frac{1}{n}\sum_{i=1}^{n} \ell(\theta_t, a_i)_i)] \tag{2}$$

$$\approx \sum_{i=1}^{n} r_{t,i} \cdot \nabla_{\phi_t} \log p(a_i;\phi_t), \tag{3}$$

where $a_i \sim p(\cdot;\phi_t)$. In Figure 1 we visualize the computational flow of this update in comparison to the standard SGD training-loop.

The following theorem should give some intuition for the relationship of GAR with the unrolled gradient loop.

Theorem 1. *The GAR update is an unbiased estimator of the meta-gradient (i.e. $\nabla_{\phi_{(t-1)}}\mathcal{L}(\theta_t)$) in the infinite batch size limit.*

Proof. Let our policy, trained with the meta-objective to maximize the GAR, be a distribution p that depends on ϕ. Further, assume that the model, and therefore the loss, depends only on ϕ through samples from p. We can thus denote the example loss $\ell(\theta,\phi)_i$ as $\ell(\theta, a_i)_i$, where $a_i \sim p(\phi)$. We consider a standard SGD update $\theta_{t+1} := \theta_t - \alpha\nabla_\theta \frac{1}{n}\sum_{i=1}^{n} \ell(\theta_t, a_i)_i$ for some given model state θ_t and meta actions $a_i \sim p(\phi)$. In the infinite batch size setting we can sample infinitely many meta actions a per batch. Thus, in the following we assume for the batch loss $l(a)$, which might depend on meta actions a, that $l(a) = \mathbb{E}_{a'\sim p(\phi_t)}[l(a')]$, for $a_i \sim p(\phi)$. This is trivially fulfilled for infinite batches of the form $\mathbb{E}_{a'\sim p(\phi_t),\ell'}[\ell'(\theta_t, a')]$. We denote the loss for the meta-gradients as $\mathcal{L}(\theta_{t+1})$. From this we can infer that the distributional gradient of the update of an algorithm trained with the GAR and the REINFORCE trick point in the same direction:

$$\nabla_{\phi_t}\mathcal{L}(\theta_{t+1})$$

Using the definition of θ_{t+1} and the infinite batch assumption.

$$= \nabla_{\phi_t}\mathcal{L}(\theta_t - \alpha\nabla_{\theta_t}\frac{1}{n}\sum_{i=1}^{n}\mathop{\mathbb{E}}_{a_i\sim p(\cdot;\phi_t)}[\ell(\theta_t, a_i)_i])$$

Apply the chain rule.

$$= -\alpha\nabla_{\theta_{t+1}}\mathcal{L}(\theta_{t+1}) \cdot \nabla^2_{\phi_t,\theta_t}\frac{1}{n}\sum_{i=1}^{n}\mathop{\mathbb{E}}_{a_i\sim p(\cdot;\phi_t)}[\ell(\theta_t, a_i)_i]$$

Now we re-arrange sums, expectations and gradients

$$= -\alpha\nabla_{\theta_{t+1}}\mathcal{L}(\theta_{t+1}) \cdot \frac{1}{n}\sum_{i=1}^{n}\nabla_{\phi_t}\mathop{\mathbb{E}}_{a_i\sim p(\cdot;\phi_t)}[\nabla_{\theta_t}\ell(\theta_t, a_i)_i]$$

Make use of the REINFORCE trick.

$$= -\alpha\nabla_{\theta_{t+1}}\mathcal{L}(\theta_{t+1}) \cdot \frac{1}{n}\sum_{i=1}^{n}\mathop{\mathbb{E}}_{a_i\sim p(\cdot;\phi_t)}[\nabla_{\theta_t}\ell(\theta_t, a_i)_i \cdot \nabla_{\phi_t}\log p(a_i;\phi_t)]$$

In the limit of the infinite batch assumption θ_{t+1} does not depend on a, since we take expectations over a and do not only sample.

$$= -\alpha \mathop{\mathbb{E}}_{a\sim p(\cdot;\phi_t)}\left[\frac{1}{n}\sum_{i=1}^{n}\langle\nabla_{\theta_t}\ell(\theta_t, a_i)_i, \nabla_{\theta_{t+1}}\mathcal{L}(\theta_{t+1}, \phi_{t+1})\rangle \cdot \nabla_{\phi_t}\log p(z;\phi_t)\right]$$

Finally we apply the definition of the GAR r.

$$= -\frac{\alpha}{n}\mathop{\mathbb{E}}_{a\sim p(\cdot;\phi_t)}\left[\sum_{i=1}^{n} r_{t,i} \cdot \nabla_{\phi_t}\log p(z;\phi_t)\right].$$

□

The above proof shows that comparing example gradients from the current step with the aggregated gradient of the next step is a bias-free estimator of the stochastic meta-gradient in the infinite batch size limit.

The GAR will only consider the impact of ϕ_t on the update generated with the last batch and will not consider the impact of ϕ_t, like noted in Algorithm 1 by the stop gradient sg.

Our method just requires computing the dot products of gradients, besides computing the gradients inside p on top of the terms which are anyways needed for an SGD loop. Similar setups where proposed before, but in the following section we also detail how to compute the GAR efficiently. With our optimized implementation, we empirically incur an increase in training time of less than 25% (a stark improvement over the direct gradient computation which had an overhead of around 540%) and a memory overhead of less than 80% (a 2× improvement) in the same setting as used for comparison in Section 3.

5 Efficiently Computing the GAR

This section details the efficient computation of the GAR, the method would work without the following strategies, but not as fast and memory-saving.

The GAR is the dot product between a batch gradient and an example gradient. In this section we assume we are given some arbitrary batch gradient g and compute the alignment of it with each example gradient of a given batch. To compute the gradient-alignment reward efficiently we use the BACKPACK package (Dangel, Kunstner, and Hennig 2020), which gives us easy access to the incoming gradients of each layer. The full gradients of a model are a concatenation of the weights of multiple layers. The dot product between two full model gradients is thus the sum of the dot products between the weights of their respective layers. Below we show how we compute gradient dot products for the three main weight types in neural networks. We refer to the batch size as n and arbitrary dimensions that depend on the model as d_i for some integer i.

In our derivations we use the per-example gradient $\partial \ell_i / \partial w$ of a weight w, which is only cleanly defined as part of the batch gradient if there is no interaction between the example computations in a batch. This is the case in most current neural networks if batch normalization is not used.

Biases Biases in neural networks are a simple vector addition of a bias vector $b \in \mathbb{R}^{d_1 \times \cdots \times d_k}$ to each hidden state x_i in a batched hidden state $x \in \mathbb{R}^{n \times d_1 \times \cdots \times d_k}$. The computation performed with a bias is $x'_i = x_i + b$. We receive the incoming gradient $\partial \ell / \partial x' \in \mathbb{R}^{n \times d_1 \times \cdots \times d_k}$ from PyTorch's autograd (Paszke et al. 2017). We simply compute $\langle (\partial \ell / \partial x')_i, g \rangle$ for all $i \in \{1, \dots, n\}$ sequentially. Biases are usually only small, thus this is not very expensive in general.

Linears Linears perform matrix multiplications between a batch of incoming hidden vectors $x \in \mathbb{R}^{n \times d_1}$ and a weight matrix $w \in \mathbb{R}^{d1 \times d2}$ to yield a new batch of hidden vectors $x' = x \cdot w \in \mathbb{R}^{n \times d_2}$. The per-example gradient for the i-th example can be computed as the outer-product $\partial \ell_i / \partial w = x_i \cdot (\partial \ell / \partial x')_i^\intercal$, where $\partial \ell / \partial x' \in \mathbb{R}^{n \times d_2}$ is computed by autograd.

Lemma 2. *The dot product between the per-example gradient $\partial \ell_i / \partial w$ and the given next batch gradient g is $(x_i^\intercal \cdot g) \cdot (\partial \ell / \partial x')_i$.*

The proofs of Lemmas 2 and 3 are given in Appendix A.

Based on Lemma 2, for a given batch we can compute the dot product with the element-wise gradient as $(x \cdot g) \cdot (\partial l / \partial x')^\intercal$. This way of computing the dot products is only marginally more expensive than a forward pass and requires much less memory than computing the products sequentially. A similar derivation for computing gradient norms was previously shown by Dangel, Kunstner, and Hennig (2020).

Convolutions Convolutions are an essential component of most neural network architectures for vision tasks. In the following we will only discuss the single channel case with implicit zero-padding and a stride of one for simplicity, but the analysis extends to many channels, other padding strategies and strides. To introduce our highly optimized implementation, we first remember the operation a convolution with a convolution matrix $K \in \mathbb{R}^{c_1 \times c_2}$ performs on an input $x \in \mathbb{R}^{n_1 \times n_2}$. Unlike previously, for this optimization it is enough to consider a single example of size $n_1 \times n_2$ instead of a batch. To simplify the problem, we assume the height and width of the kernel, c_1 and c_2, to be odd. Since we use zero-padding we assume in the following calculations that out of bounds indexes yield zeros, or equivalently that x has an all zero frame of widths $\lfloor \frac{c_1}{2} \rfloor$ and $\lfloor \frac{c_2}{2} \rfloor$ in dimension 2 and 3. We can now define the convolution operator as

$$\begin{aligned} x'_{i_1,i_2} &= (x * K)_{i_1,i_2} \\ &= \sum_{j_1=1}^{c_1} \sum_{j_2=1}^{c_2} K_{j_1,j_2} \cdot x_{i_1+j_1-\lfloor \frac{c_1}{2} \rfloor, i_2+j_2-\lfloor \frac{c_2}{2} \rfloor}, \end{aligned}$$

where we denote indexes as function arguments. Further we recall that the gradient of a convolution towards its weight $\frac{\partial \ell}{\partial K}$ can be defined as

$$\frac{\partial \ell}{\partial K_{j_1,j_2}} = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \frac{\partial \ell}{\partial x'_{i_1,i_2}} \cdot x_{i_1+j_1-\lfloor \frac{c_1}{2} \rfloor, i_2+j_2-\lfloor \frac{c_2}{2} \rfloor}.$$

Lemma 3. *Given the above assumptions on the convolution function, we have that the dot product of a given matrix $g \in \mathbb{R}^{c_1 \times c_2}$ with each per-example gradient can be written as $\langle g, \frac{\partial \ell_i}{\partial K} \rangle = \sum_{i_1=1}^{n_1} \sum_{i_2=1}^{n_2} \frac{\partial \ell}{\partial x'_{i_1,i_2}} \cdot (x * g)_{i_1,i_2}$*

The new form of the dot product simply consists of an element-wise product between the incoming gradient and the result of using g instead of K in the convolution. Thus, we can compute the dot product between g and the gradient of each individual example in a batch very cheaply. The costs are the same as the forward pass through the convolution except for the final dot product.

6 Experiments

We performed experiments in two different setups. We will first detail an interpretable and easy-to-reproduce toy experiment, than we will detail the application of the GAR of inloop meta learning of augmentations.

Illustrative Example: Batch Sampling Distribution

We perform a motivational experiment on a small toy task. We split the CIFAR-10 dataset into 10 equally sized parts and on the zeroth part we replace all labels with labels drawn uniformly at random. Therefore, on split 0 most labels are wrong. Things learned from split 0 will, thus, generalize badly to other splits. For each example in an SGD batch, we first sample the split from a learned distribution and then uniformly sample from within that split. We make experiments with three networks: a fully-connected network with a single hidden layer of size 200 (FC), the same fully-connected network but with BatchNorm (Ioffe and Szegedy 2015) (FC$_{BN}$) and a small CNN (CNN$_{BN}$) with a single convolution of size 3 from the input to a single channel, followed by BatchNorm and a linear layer. In all networks we use ReLU activations between layers. We use a batch size of 1000, train for 10 epochs and apply an SGD optimizer with Nesterov momentum of 0.9, a .0005 L$_2$ regularization and a fixed learning rate of 0.1. As for the meta optimization, we parameterize the distribution over splits by an "inverted" softmax distribution generated from learned logits s, that is $p(s)_i \sim 1 - \mathrm{softmax}(s)_i$. This is useful compared to other distributions, since we saw that, while softmax has a bias towards a single winner, this has a tendency towards a single loser. To train s online with GAR we use policy gradient. We aggregate gradients over 10 steps, normalize the rewards for each step and use Adam (Kingma and Ba 2015) with a learning rate of 0.1. To compare our method fairly. We compare it to NSLR, as proposed in section 4, which is much simpler, but still novel. While there is only one NSLR reward per step, this baseline yields a bias free estimator of the true stochastic meta-gradient. We performed all experiments with 10 different seeds. In Figure 2 we show the allocation of $p(s)$ over time for GAR and the NSLR, as well as training losses over time, the usage of each bucket, for the fully-connected network with batch norm. We can see that GAR presses down the noisy category with little variance in Figure 2b, while the baseline does not succeed in omitting the noisy data, as we see in Figure 2a. In Figure 2c we see how this impacts training losses of the two setups. We could see similar results for the other setups and show the area under the curve (AUC) of the split probability (the usage) for these in Table 1. An optimal method would show a low AUC for the noisy split, but a high AUC for the other splits. It would, thus omit training on noisy data, but not omit training on any of the non-noisy splits. To experiment with different networks, setups or add noise to the images instead of the labels, we refer to our public colab notebook[1] generating these experiments with no setup.

Net	Method	Noisy Split	Other Splits
FC$_{BN}$	GAR	0.61	0.93
	NSLR	0.96	0.89
FC	GAR	0.65	0.93
	NSLR	0.86	0.90
CNN$_{BN}$	GAR	0.69	0.92
	NSLR	0.92	0.90

Table 1: Average AUC for the split probability over 10 epochs of (non-)noisy splits for different architectures.

	PBA	Fast AA	AA	RA	OLA (WL)	OLA (RA)
CIFAR-10						
UA Baseline	-	-	-	-	97.61 ± 0.16	97.51 ± 0.18
Method	97.4	97.3	97.4	97.3	97.39 ± 0.15	97.56 ± 0.07
CIFAR-100						
UA Baseline	-	-	-	-	83.20 ± 0.34	83.40 ± 0.09
Method	83.3	82.7	82.9	83.3	84.30 ± 0.40	83.54 ± 0.14

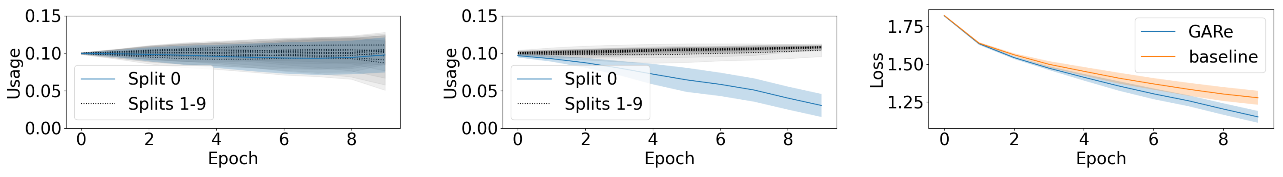
Table 2: Results of the OLA experiments. The results are test accuracies and the 95% confidence interval is noted with $\pm$.

Online-Learned Augmentation-Strategy

The choice of image augmentation was shown to have enough impact to improve performance considerably (Cubuk et al. 2019). We apply the GAR to learn the augmentation policy online. We refer to this method as *Online-Learned Augmentation-Strategy (OLA)*. We use a setup inspired by RandAugment (Cubuk et al. 2020). We have a set of augmentations $\mathcal{A}$ and for each image we sample a learned number $r \in \{1, \ldots, 4\}$ of augmentations $a_1, \ldots, a_r \in \mathcal{A}$ uniformly without replacement. r itself is sampled from a learned distribution $p(r) = \mathrm{softmax}(l^{(r)})$. We apply each augmentation a_i in sampling order to the example image, each with a sampled strength $k_{a_i} \in \{0, \ldots, 30\}$. k_{a_i} is sampled from the distribution $p(k_{a_i}) = \mathrm{softmax}(l^{(a)}_{a_i})$, which depends on the applied augmentation a_i. Each sampled augmentation a_i is actually applied with a learned probability $p(d_{a_i}) = \sigma(l^{(d)}_{a_i})$. A sampled augmentation might thus not be applied after all. In the above all logits $l^{(r)} \in \mathbb{R}^4$, $l^{(a)} \in \mathbb{R}^{|\mathcal{A}| \times 31}$ and $l^{(d)} \in \mathbb{R}^{|\mathcal{A}|}$ are learned weights and initialized to zero. We outline the augmentation sampling process in Algorithm 2.

We perform our evaluations on CIFAR-10 and CIFAR-100 using a WideResnet-28-10 (Zagoruyko and Komodakis 2016) and follow the setup of Cubuk et al. (2020) in detail, but we use a larger batch size of 256. We keep the number of epochs stable, though, such that we do not have an unfair advantage. To prevent the augmentation distribution from collapsing we interleave steps with learned augmentations with steps that do not use augmentations. Such that we consider alignments of augmented examples with a non-augmented batch. For maximal efficiency we use the gradients computed on non-augmented batches not only to compute the GAR, but treat them as normal steps in the main training and update the neural network with them. Like before we

[1]https://bit.ly/34K7aAT



(a) (b) (c)

Figure 2: Average split distribution of the NSLR baseline (a) and GAR (b) across epochs. Training losses are given in (c).

Algorithm 2 OLA Augmentation Sampling Procedure

```
 1: receive sample image $x$
 2: sample the number of augmentations $r \sim p(r)$
 3: sample the augmentations to apply $a_1, \ldots, a_r$ uniformly
    at random without replacement from $\mathcal{A}$
 4: for $i \in \{1, \ldots, r\}$ do
 5:     sample keep indicator $d_{a_i} \sim p(d_{a_i})$
 6:     if $d_{a_i}$ then
 7:         sample strength $k_{a_i} \sim p(k_{a_i})$
 8:         apply the augmentation $x \leftarrow a_i(x, k)$
 9:     end if
10: end for
```

Algorithm 3 UA Augmentation Sampling Procedure

```
1: receive sample image $x$
2: sample the augmentations to apply $a_1$ and $a_2$ uniformly
   at random without replacement from $\mathcal{A}$
3: for $i \in \{1, 2\}$ do
4:     sample keep indicator $d \sim \text{Bern}(0.5)$
5:     if $d$ then
6:         sample strength $k$ uniformly at random from
           $\{0, \ldots, 30\}$
7:         apply the augmentation $x \leftarrow a_i(x, k)$
8:     end if
9: end for
```

use Adam for the meta optimization. We set the learning rate of Adam to 0.1, normalize the rewards in each batch and do not aggregate meta gradients.

We compare OLA with a set of common augmentation strategies, most of which have to pre-trained. We compare with Population based Augmentation (PBA; Ho et al. 2019), AutoAugment (AA; Cubuk et al. 2019), Fast AA (Lim et al. 2019) and RandAugment (RA; Cubuk et al. 2020). So far in the literature little attention was given to the search space. Most previous work use slightly different search spaces. PBA, AA and RandAugment all have slightly different search spaces for example. In our experiments we found the search space choice to be important. We reimplemented UniformAugment (UA; LingChen et al. 2020) and found that surprisingly in our setup we could considerably improve the performance of our reimplementation of UA depending on the search space, something we were not able to do for RA. Therefore, unlike previous work, we provide our reimplementation of UA as an additional baseline, since it was evaluated under the exact same settings. Algorithm 3 outlines our reimplementation of UA. We made two main changes: We sample strengths k from a range of integer values instead of a real-valued range, to align with our comparisons, and we sample the augmentations without replacement, which makes the set of applied augmentations more diverse. Other than that, we removed the double sampling of the keep probability and replaced with an equivalent single sampling.

In Table 2 we show the average test accuracies over 5 runs of our method and the UA baselines with confidence bounds and the comparisons from literature. We evaluated our method on two different search spaces, which we denote in parentheses for our evaluations. The search space RA is equivalent to the search space used for RandAugment, while WideLong (WL) is a search space that includes more extreme strength settings and more augmentations, see Table 3. While our method and its baseline perform well in comparison to previous methods, the comparison we want to focus on is the comparison with our reimplementation of UA, since we share all setup with it. For CIFAR-10 one can see here, that we are performing similar to the strong baselines for both search spaces. For CIFAR-100 we outperform the baselines in both cases by at least the 95% confidence interval. Note, that unlike previous methods we do not pre-train or do hyper-parameter search on a per-dataset basis. Our results point out that over different setups our method works either comparable or better than previous methods and the UA baseline, while other methods like AA, are expensively pre-trained for each dataset. Also worthy of mention is how well the UA baseline performs, even for the larger search space, compared to learned methods.

7 Conclusion

We presented a way of adapting meta parameters ϕ online during SGD training of the elementary network parameters θ that is efficient in terms of data, memory and time and applies to optimizing non-differentiable distributions during training. Key to our approach is the Gradient Alignment Reward, which allows using reinforcement learning with unique per-sample rewards. We showed its benefits on an interpretable toy task and a real world task. This method has many potential future applications like large scale learned curricula or neural architecture search.

Acknowledgements The authors acknowledge funding by the Robert Bosch GmbH.

References

Bengio, Y.; LeCun, Y.; et al. 2007. Scaling learning algorithms towards AI. *Large-scale kernel machines* 34(5): 1–41.

Chen, P.; Liao, B. B.; Chen, G.; and Zhang, S. 2019. Understanding and Utilizing Deep Neural Networks Trained with Noisy Labels. In *Proc. of ICML'19*, 1062–1070.

Cubuk, E. D.; Zoph, B.; Mané, D.; Vasudevan, V.; and Le, Q. V. 2019. AutoAugment: Learning Augmentation Strategies From Data. In *Proc. of CVPR'19*, 113–123.

Cubuk, E. D.; Zoph, B.; Shlens, J.; and Le, Q. 2020. RandAugment: Practical Automated Data Augmentation with a Reduced Search Space. In *Proc. of NeurIPS'20*.

Dangel; Kunstner, F.; and Hennig, P. 2020. BackPACK: Packing more into Backprop. In *Proc. of ICLR'20*.

DeVries, T.; and Taylor, G. W. 2017. Improved regularization of convolutional neural networks with cutout. *arXiv:1708.04552 [cs.CV]* .

Graves, A.; Bellemare, M. G.; Menick, J.; Munos, R.; and Kavukcuoglu, K. 2017. Automated curriculum learning for neural networks. In *Proc. of ICML'17*, 1311–1320.

Grefenstette, E.; Amos, B.; Yarats, D.; Htut, P. M.; Molchanov, A.; Meier, F.; Kiela, D.; Cho, K.; and Chintala, S. 2019. Generalized Inner Loop Meta-Learning. *arxiv:1910.01727 [cs.LG]* .

Ho, D.; Liang, E.; Chen, X.; Stoica, I.; and Abbeel, P. 2019. Population based augmentation: Efficient learning of augmentation policy schedules. In *Proc. of ICML'19*, 2731–2741.

Ioffe, S.; and Szegedy, C. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proc. of ICML'15*.

Kingma, D.; and Ba, J. 2015. Adam: A Method for Stochastic Optimization. In *Proc. of ICLR'15*.

Krizhevsky, A.; Hinton, G.; et al. 2009. Learning multiple layers of features from tiny images .

Lim, S.; Kim, I.; Kim, T.; Kim, C.; and Kim, S. 2019. Fast AutoAugment. In *Proc. of NeurIPS'19*, 6665–6675.

LingChen, T. C.; Khonsari, A.; Lashkari, A.; Nazari, M. R.; Sambee, J. S.; and Nascimento, M. A. 2020. UniformAugment: A Search-free Probabilistic Data Augmentation Approach. *arxiv:2003.14348 [cs.CV]* .

Liu, H.; Simonyan, K.; and Yang, Y. 2019. DARTS: Differentiable Architecture Search. In *Proc. of ICLR'19*.

Luketina, J.; Berglund, M.; Greff, K.; and Raiko, T. 2016. Scalable Gradient-Based Tuning of Continuous Regularization Hyperparameters. In *Proc. of ICML'16*, 2952–2960.

Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; and Lerer, A. 2017. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop*.

Wang, X.; Pham, H.; Michel, P.; Anastasopoulos, A.; Carbonell, J.; and Neubig, G. 2020. Optimizing Data Usage via Differentiable Rewards. In *Proc. of ICML'20*, 9983–9995.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8(3-4): 229–256.

Zagoruyko, S.; and Komodakis, N. 2016. Wide residual networks. In *Proc. of BMCV'16*.

A Proofs of Lemmas 2 and 3

Proof of Lemma 2.

$$\begin{aligned}\langle \partial\ell_i/\partial w, g\rangle &= \sum_{k_1=1}^{d_1}\sum_{k_2=1}^{d_2}(\partial\ell_i/\partial w)_{k_1,k_2}\cdot g_{k_1,k_2}\\ &= \mathrm{Tr}((\partial\ell_i/\partial w)^\intercal\cdot g)\\ &= \mathrm{Tr}((x_i\cdot(\partial\ell/\partial x')_i^\intercal)^\intercal\cdot g)\\ &= \mathrm{Tr}(((\partial\ell/\partial x')_i\cdot x_i^\intercal)\cdot g)\\ &= \mathrm{Tr}((\partial\ell/\partial x')_i\cdot(x_i^\intercal\cdot g))\\ &= \langle(\partial\ell/\partial x')_i,(x_i^\intercal\cdot g)\rangle.\end{aligned}$$

□

Proof of Lemma 3.

$$\begin{aligned}&\sum_{j_1=1}^{c_1}\sum_{j_2=1}^{c_2}\frac{\partial\ell}{\partial K_{j_1,j_2}}\cdot g_{j_1,j_2}\\ &=\sum_{j_1=1}^{c_1}\sum_{j_2=1}^{c_2}\sum_{i_1=1}^{n_1}\sum_{i_2=1}^{n_2}\frac{\partial\ell}{\partial {x'}_{i_1,i_2}}\cdot x_{i_1+j_1-\lfloor\frac{c_1}{2}\rfloor,i_2+j_2-\frac{c_2}{2}}\cdot g_{j_1,j_2}\\ &=\sum_{i_1=1}^{n_1}\sum_{i_2=1}^{n_2}\frac{\partial\ell}{\partial {x'}_{i_1,i_2}}\sum_{j_1=1}^{c_1}\sum_{j_2=1}^{c_2}x_{i_1+j_1-\lfloor\frac{c_1}{2}\rfloor,i_2+j_2-\lfloor\frac{c_2}{2}\rfloor}\cdot g_{j_1,j_2}\\ &=\sum_{i_1=1}^{n_1}\sum_{i_2=1}^{n_2}\frac{\partial\ell}{\partial {x'}_{i_1,i_2}}\cdot(x*g)_{i_1,i_2}.\end{aligned}$$

□

B WideLong Search Space

PIL operation	range	PIL operation	range
identity	-	auto_contrast	-
equalize	0.01 - 2.0	rotate	$-135°$ - $135°$
solarize	0 - 256	color	0.01 - 2.0
posterize	0.01 - 2.0	contrast	0.01 - 2.
brightness	2 - 8	sharpness	0.01 - 2.0
shear_x	0.0 - 0.99	shear_y	0.0 - 0.99
translate_x	0 - 32	translate_y	0 - 32
blur	-	invert	-
flip_lr	-	flip_ud	-
cutout	0 - 19		

Table 3: The WideLong (WL) search space. All methods are defined as part of Pillow (https://github.com/python-pillow/Pillow), as part of ImageEnhance, ImageOps or as image attribute, besides cutout (DeVries and Taylor 2017).