AutoLoss: Learning Discrete Schedule for Alternate Optimization

Anonymous Author(s)
Affiliation
Address
email

Abstract

Many machine learning problems involve iteratively and alternately optimizing different task objectives with respect to different sets of parameters. Appropriately scheduling the optimization of a task objective or a set of parameters is usually crucial to the quality of convergence. In this paper, we present *AutoLoss*, a meta-learning framework that automatically learns and determines the optimization schedule. AutoLoss provides a generic way to represent and learn the discrete optimization schedule from metadata, allows for a dynamic and data-driven schedule in ML problems that involve alternating updates of different parameters or from different loss objectives. we apply AutoLoss on three ML tasks: d-ary quadratic regression, classification using a multi-layer perceptron (MLP), and image generation using GANs, and show that, on all three tasks, the AutoLoss controller is able to capture the distribution of better optimization schedules that result in higher quality of convergence on all three tasks. The trained AutoLoss controller is generalizable – it can guide and improve the learning of a new task model with different specifications, or on different datasets.

1 Introduction

Many machine learning (ML) problems involve iterative alternate optimization of different objectives $\{\ell_m\}_{m=1}^M$ w.r.t different subsets of parameters $\{\boldsymbol{\theta}_n\}_{n=1}^N$ until a global concensus is reached. For instances, in training generative adversarial networks (GANs) [15], parameters of the generator and the discriminator are alternately updated to an equilibrium; in a typical expectation-maximization (EM) framework [24], one usually has to alternate variable steps of computation to obtain an estimation of the expectation, and then variable steps of optimization for model parameters, until the objective (e.g. likelihood) is maximized. In these processes, one needs to determine which objective ℓ_m and which set of parameters $\boldsymbol{\theta}_n$ to choose at each step, and subsequently, how many iteration steps to perform for the subproblem $\min_{\boldsymbol{\theta}} \ell_t$. We refer to this as determining an *optimization schedule* (or update schedule).

While extensive research has been done to develop either better optimization algorithms or update rules [19, 5, 12, 30, 31], how to create a better optimization schedule has remained less studied. When the optimization target is complex (e.g. non-convex or combinatorial) and the parameters to be optimized are high-dimensional, the optimization schedule can directly impact the quality of convergence. However, we hypothesize that the optimization schedule is learnable in a data-driven way, with the following empirical evidences: (1) We observe that the training processes of many ML models are sensitive to the update schedule. For examples, in GANs, the updates for the generator and the discriminator are carefully balanced to avoid otherwise model collapse or gradient vanishing [15, 27]; In maximum likelihood estimation (MLE) with EM and stochastic variational inference [18], the number of steps for variational updates at E-step and parameter updates at M-step may result in a different efficiencies or quality of convergence [33, 6], etc. (2) Consider in solving many multi-task learning or regularizer-augmented tasks, where the global objective $\mathcal{L}$

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

is a combination of multiple task-specific objective functions; it is usually desired to control the importance of each objective with a weight λ_m: $\mathcal{L} = \sum_{m=1}^{M} \lambda_m \ell_m$. Different values of $\{\lambda_m\}_{m=1}^{M}$ result in different (local) optima. It drives us to think that, in order to achieve best downstream task performance, different losses shall not be treated equally, and the solution that yields the best results requires optimizing every ℓ_m to different extents. (3) Previous research and practice have suggested there exist some optimization schedules that are highly possible to produce better convergence results than random ones, e.g. some literatures [2, 29, 11] suggest that keeping the steps of updating the generator and discriminator of GANs at $K : 1(K > 1)$ leads to faster and more stable training.

Based on the hypothesis, in this paper, we develop a generic meta-learning framework, called *AutoLoss*, to automatically determine the optimization schedule in iterative and alternate optimization processes. AutoLoss introduces a parametric controller in attached to an alternate optimization task. The controller is designed to capture the relations between the past history and the current state of the optimization process, and the next step of decision on the optimization schedule – it takes as input a set of specific features describing the current status, and makes a decision for the next step of optimization by answering the question: which set of objective terms from $\{\ell_m\}_{m=1}^{M}$ to optimize, and which set of parameters from $\{\boldsymbol{\theta}_m\}_{n=1}^{N}$ to update. The controller is trained via policy gradient to maximize the eventual outcome of the optimization (e.g. downstream task performance). Once trained, it can guide the learning of task model by providing scheduling information: it first predicts a set of losses and parameters, and instructs the task model to only perform one step of optimization over them, and eventually helps the task model to achieve a higher quality of convergence by providing better schedules.

To evaluate the effectiveness of AutoLoss, we instantiate it to model the learning process of three typical ML tasks: d-ary quadratic regression, classification using a multi-layer perceptron (MLP), and image generation using GANs. We propose an effective set of features and reward functions that are suitable for the controllers' learning and decisions. We show that, on all three tasks, the AutoLoss controller is able to capture the distribution of better optimization schedules that result in higher quality of convergence on the corresponding task than strong baselines. For example, on quadratic regression with L1 regularization, it learns to detect the potential risk of overfitting, and incorporates L1 regularization when necessary, achieving a better converged results that can hardly be achieved by optimizing a linear combination of objective terms; on GANs, the AutoLoss controller learns to balance the training of generator and discriminator dynamically, and report both faster per-epoch convergence and better quality of generators after convergence, compared to fixed heuristic-driven schedules.

In summary, we made the following contributions in this paper: (1) We present a unified formulation for iterative and alternate optimization processes, based on which, we develop AutoLoss, a generic framework to learn the discrete optimization schedule of such processes using reinforcement learning (RL). To our knowledge, this is the first framework that tries to learn the optimization schedule in a data-driven way. (2) We instantiate AutoLoss on three ML tasks: d-ary regression, MLP classification, and GANs; We propose a novel set of features and reward functions for the AutoLoss controller to capture better schedules; (3) We empirically demonstrate AutoLoss' efficacy: it delivers higher quality of convergence for all three tasks on synthetic and real dataset than strong baselines; the trained AutoLoss controller is generalizable – it can guide and improve the training of a new task model with different specifications, or on a different dataset.

2 Related Work

Alternate Optimization. Many ML models are trained using algorithms with iterative and alternate workflows, such as EM [24], stochastic gradient descent (SGD) [7], coordinate descent [35], etc. AutoLoss can improve this kind of process by learning a controller in a data-driven way, and figuring out a better scheduling of updates using this controller, as long as the schedule does affect the optimization goal. In this paper, we focus mostly on optimization problem, but note AutoLoss is generic to alternate processes that involve non-optimization subtasks, e.g. sampling methods [16, 21].

Meta learning. Meta learning [1, 22, 32, 14, 10] has drawn considerable interest from the community, and has been recently applied to improve the optimization of ML models [28, 20, 5, 13]. Among these works, the closest to ours are [20, 5, 13]. In [20], a method called learning to optimize is proposed to directly predict the values of gradients at each step of SGD. Since the values are continuous and might be high-dimensional, directly regressing them might be difficult. Also, the learned gradient predictor is not transferable to other models or tasks. Differently, [5] proposes a domain specific

language (DSL) and learns a gradient update rule following the DSL. The learned rules prove perform better than manually designed ones, and is generalizable. AutoLoss is different from this line of works – instead of learning to generate better values of updates (gradients) than human-designed ones, AutoLoss focuses on producing better scheduling of the updates. Therefore AutoLoss can model another class of problems such as scheduling the generator and discriminator training of a GAN, or even go beyond optimization problems as we have discussed. In [13], a learning to teach framework is proposed that a teacher model that captures different aspects of metadata can be trained to guide the learning of student models. AutoLoss can be thought as an instantiation of learning to teach that the teacher model produces better scheduler for the alternate optimization process.

Also of note is another line of works that apply RL to predict discrete variables, such as device placement [23], neural architecture [3, 36], etc. AutoLoss' controller is trained similarly [25] to produce sequential and discrete predictions, though addressing different problems.

3 AutoLoss

Background. In most machine learning tasks, we are given observed dataset $\mathcal{D}$, and we aim to minimize an objective function $\mathcal{L}(\mathcal{D}; \mathbf{\Theta})$ of the data $\mathcal{D}$, with respect to $\mathbf{\Theta}$, which is the parameters of the model that we used to characterize the data. Solving this minimization problem involves finding the optimal value of $\mathbf{\Theta}$ (denoted as $\mathbf{\Theta}^*$), which we usually resort to either closed form solutions (if possible), or a variety of de facto optimization methods [8]. In the rest of the paper, we will focus on two typical classes of optimization workflows where the AutoLoss framework is built upon: *iterative* and *alternate* optimizations, which many modern ML models solvers would fall into.

Iterative optimization methods look for the optimal parameter $\mathbf{\Theta}^*$ in an *iterative-convergent* way, by repeatedly updating the parameters $\mathbf{\Theta}$ until certain stopping criteria is reached. Specifically, at iteration t, the parameters $\mathbf{\Theta}$ are updated from $\mathbf{\Theta}^{(t)}$ to $\mathbf{\Theta}^{(t+1)}$ following the update equation $\mathbf{\Theta}^{(t+1)} = \mathbf{\Theta}^{(t)} + \epsilon \cdot \Delta_{\mathcal{L}}(\mathcal{D}^{(t)}; \mathbf{\Theta}^{(t)})$, where we denote $\Delta_{\mathcal{L}}$ as the update function that calculates update values of $\mathbf{\Theta}$ depending on $\mathcal{L}$, $\mathcal{D}^{(t)} \subseteq \mathcal{D}$ as a subset of $\mathcal{D}$ used at iteration t and ϵ a scaled factor. Many widely-adopted algorithms fall into this family, such as SGD [7] and subgradient methods [9]. For instance, in the case for SGD, $\Delta_{\mathcal{L}}$ reduces to deriving the gradient updates $\nabla\mathbf{\Theta}$ (we skip the optional transformation steps such as momentum or projection for clarity), $D^{(t)}$ is a stochastic batch, and ϵ is the learning rate.

To generalize this formulation to describe alternate optimization, we note that in a more realistic scenario, we usually have the objective function $\mathcal{L}$ being composed by multiple different optimization targets: $\mathcal{L} = \{\ell_m\}_m^M$, and we want the optimal solution $\mathbf{\Theta}^*$ to minimize a certain combination of them. For example, when fitting a regression model with mean square error (MSE), we usually prefer to appending an L1 loss to obtain sparsity; in this case, the objective function $\mathcal{L}$ is written as a linear combination of two separate terms. Similarly, the parameter $\mathbf{\Theta}$ to be optimized in many cases is also composable, e.g. when the used model has multiple components with independent sets of parameters. If we decompose the parameter $\mathbf{\Theta}$ into N vectors as $\mathbf{\Theta} = \{\boldsymbol{\theta}_n\}_{n=1}^N$, an alternate optimization (in our definition) contains multiple steps; at each step t it involves choosing a set of pairs $\{(\ell, \boldsymbol{\theta})\}$ of objective function and parameter then updating each $\boldsymbol{\theta}$ w.r.t. ℓ for each pair.

Further, we note that many ML optimization tasks in practical are both iterative and alternate. In particular, at each step t during the alternate optimization, we may have no access to a closed form solution for the subproblem $\min_{\boldsymbol{\theta}} \ell$. We instead update $\boldsymbol{\theta}$ iteratively with multiple sub-steps. To see a concrete example, consider the training process of GANs: one iteratively updates the parameters of generator G or discriminator D by applying the gradients of either G or D w.r.t. their optimization target using a stochastic batch of data.

A unified formulation. It is therefore natural to present iterative and alternate optimization in a unified workflow, as in the following equation:

$$\text{for } t = 1 \to T, \quad \boldsymbol{\theta}_n^{(t+1)} = \boldsymbol{\theta}_n^{(t)} + \epsilon \cdot \sum_{m=1}^{M} y_{n,m}^{(t)} \cdot \Delta_{\ell_m}^n, n = 1 \sim N \tag{1}$$

$y_{n,m}^{(t)}$ is a binary variable which should be set to 1 if we want to update θ_n w.r.t. ℓ_m at step t and 0 otherwise. $\Delta_{\ell_m}^n$ is denoted as update values of θ_n w.r.t. ℓ_m. Eq. 1 reduces to the vanilla form of iterative optimization when $y_{n,m}^{(t)} = 1, \forall t, m, n$.

AutoLoss. Give the formulation in Eq. 1, our goal is to generate values of $y_{n,m}^{(t)}$ for each n, m at each step t, in order to maximize the downstream task performance. We introduce a meta model to decide which loss to optimize and what parameters to update at each step of the optimization. To distinguish from the model used in the downstream task, we will call the meta model as a *controller* whereas the other the *task model*. We expect the controller to learn during its exploration of the optimization process, and is able to make decisions on how to update once sufficient knowledge has been accumulated. Specifically, we let the controller make sequential decision at each step t; it scans through the past history and the current states of the process (presented as a K dimensional feature vector $\boldsymbol{X}^{(t)} \in \mathbb{R}^K$), and predicts an $M \times N$ matrix $\boldsymbol{Y}^{(t)} = [[y_{1,1}^{(t)}, \cdots, y_{1,N}^{(t)}], \cdots, [y_{M,1}^{(t)}, \cdots, y_{M,N}^{(t)}]] \in \{0,1\}^{M \times N}$.

Hence, we model our controller as a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\phi})$ parameterized by $\boldsymbol{\phi}$[1], where we use $\boldsymbol{y}$ and $\boldsymbol{x}$ to denote the $(M \times N)$-dim decision variable and K-dim feature variable, respectively. At each step t, we sample a binary decision matrix $\boldsymbol{Y}^{(t)} \sim p(\boldsymbol{y}|\boldsymbol{x} = \boldsymbol{X}^{(t)}; \boldsymbol{\phi})$, and perform one step of updates following Eq. 1 and $\boldsymbol{Y}^{(t)}$. The parameters of the controller ϕ is trained to maximize the performance of the optimization task given its sampled sequences of decisions within T steps. If we notate $\mathcal{Y} = \{\boldsymbol{Y}^{(t)}\}_{t=1}^{T}$, we accordingly formulate the objective as

$$J(\phi) = \mathbb{E}_{\mathcal{Y} \sim p(\boldsymbol{y}|\boldsymbol{x}; \boldsymbol{\phi})} \big[R(\mathcal{Y}) | \mathcal{L}, \boldsymbol{\Theta} \big], \tag{2}$$

where we bring in $R(\cdot)$ as a reward function evaluates the final performance of the optimization task led by the sequential decisions $\mathcal{Y}$.

Since the decision process involves sampling, which is a non-differentiable operation, we learn the parameters ϕ using REINFORCE algorithm [34], where the unbiased policy gradients at each updating step h of the controller are estimated by sampling S sequences of decisions $\{\mathcal{Y}_s\}_{s=1}^{S}$ and compute

$$\nabla_\phi J(\phi) = \frac{1}{S} \sum_{s=1}^{S} \big[(R(\mathcal{Y}_s) - B) \cdot \nabla_\phi \sum_{t=1}^{T} \log p(\boldsymbol{Y}_s^{(t)} | \boldsymbol{X}_s^{(t)}; \boldsymbol{\phi}) | \mathcal{L}, \boldsymbol{\Theta} \big], \tag{3}$$

where $\boldsymbol{Y}_s^{(t)}$ is the decision at step t in $\mathcal{Y}_s$. When T is large, the long-range exploration of the controller results in high variance in $\nabla_\phi J(\phi)$. To reduce the variance, we hence introduce a baseline term B in Eq. 3 to stabilize the training [26], where B is defined as a moving average of received reward: $B^{(h+1)} \leftarrow \eta B^{(h)} + (1 - \eta) R^{(h)}$ with η as a decay factor. Whenever applicable, the final reward $R(\mathcal{Y}_s) - B$ is clipped to a given range to avoid exploding or vanishing gradients.

4 Applications

We next apply the AutoLoss framework to three specific ML tasks: d-ary quadratic regression with L1 regularization, classification using a two-layer MLP, and image generation using GANs.

d-ary quadratic regression with L1 regularization. Given training data $\mathcal{D} = \{\boldsymbol{u}_p, v_p\}_{p=1}^{P}, \boldsymbol{u}_p \in \mathbb{R}^d, v \in \mathbb{R}$, we try to fit the data using a d-ary quadratic model $f : \mathbb{R}^d \to \mathbb{R}$ in the form of $f(\boldsymbol{u}; \boldsymbol{\Theta}) = \boldsymbol{u}^\top \boldsymbol{A} \boldsymbol{u} + \boldsymbol{b}^\top \boldsymbol{u} + c$ with parameters $\boldsymbol{\Theta} = \{\boldsymbol{A}, \boldsymbol{b}, c\}$, where the parameters can be optimized via minimizing the MSE $\ell_1(\boldsymbol{\Theta}) = \mathbb{E}_{(\boldsymbol{u},v) \in D}[f(\boldsymbol{u}) - v]^2$. In the case where the training data were generated by a linear model with (Gaussian) noise, fitting them using a higher-order model is prone to overfitting, and adding an L1 regularization term $\ell_2 = \|\boldsymbol{\Theta}\|_1$ would help alleviate the problem. To find $\boldsymbol{\Theta}^*$, in a traditional setting, one usually minimizes a linear combination of ℓ_1 and ℓ_2 as $\ell_1 + \lambda \ell_2$, where λ is treated as a hyperparameter yet to be determined by other hyperparameter search techniques. This problem can be solved using many iterative optimization methods, e.g. SGD.

To fit this problem into the AutoLoss framework as in Eq. 1, we define $\mathcal{L} = \{\ell_1, \ell_2\}$, $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_1\}, \boldsymbol{\theta}_1 = \{\boldsymbol{A}, \boldsymbol{b}, c\}$, i.e. we consider all parameter as a single set ($N = 1$). To further simplify the problem, we restrict the action space of the controller: at each step, it chooses one of ℓ_1, ℓ_2 to optimize. This reduces the action space from 2^M to M ($M = 2$), and simplifies the controller to have a Bernoulli output which we can sample the decisions from. To train the controller parameter $\boldsymbol{\phi}$, we employ Eq. 3, and set $S = 1$ and T as the maximum number of training steps for an training instance, i.e. the controller receives a reward upon the completion of a training experiment, and updates the parameters

[1]The other alternate is to condition the decision at the t step on the decision made at the $t - 1$ step, though we choose a simpler one to highlight the generic idea behind AutoLoss.

ϕ once per experiment. For the d-ary regression, we define $\boldsymbol{X}^{(t)}$ as a concatenation of the following features in order the capture the current optimization state and the past history: (1) *training progress*: the current percentile progress of training t/T; (2) *gradient magnitude*: An M-dimensional vector where the mth entry is the normalized L2 norm of the stochastic gradients of ℓ_m over $\boldsymbol{\Theta}$: $\frac{\|\nabla_{\boldsymbol{\Theta}}\ell_m\|_2}{\sqrt{\dim(\boldsymbol{\Theta})}}$. To avoid evaluating all $\nabla_{\boldsymbol{\Theta}}\ell_m$ every step for each m, we maintain a history of the values, i.e. every step when the controller made a decision and descended the parameters using $\nabla_{\boldsymbol{\Theta}}\ell_m$, we update the corresponded entry in the history vector, and only use the latest vector as the feature for next decision; (3) *loss values*: an M-dimensional vector $[\ell_1, \ell_2]$ that contains the loss values of each ℓ_m. We similarly maintain a history vector and only use the most recent values to avoid repeating evaluating every loss at every step; (4) *validation metrics*: the MSE error ℓ_1 evaluated using $\boldsymbol{\Theta}^{(t)}$ on a validation dataset, the exponential moving average of it, and the exponential moving average of the first-order difference of it. Intuitively, we instantiate the reward function as $R = \frac{C}{\ell_1(\boldsymbol{\Theta}^{(T)})}$, where C is a contant and $\ell_1(\boldsymbol{\Theta}^{(T)})$ is the MSE evaluated using the converged parameters $\boldsymbol{\Theta}^{(T)}$ on the validation dataset. Hence, the controller obtains a larger reward if the task model achieves a lower MSE.

MLP classifier. We apply AutoLoss in training a binary multi-layer percepton (MLP) classifier[2] $f(\boldsymbol{\Theta}) : \mathbb{R}^d \rightarrow \{0, 1\}$ with ReLU nonlinearity, which is a non-convex model and recognized to be highly prone to overfitting. We materialize $\boldsymbol{\Theta}$ as a single set of parameters of the MLP, and $\mathcal{L} = \{\ell_1, \ell_2\}$ with ℓ_1 as a binary cross entropy loss, $\ell_2 = \|\boldsymbol{\Theta}\|_1$. We similarly limit the action space of controller to be binary – it chooses either ℓ_1 or ℓ_2 at each step t. We use the same set of features as for d-ary regression, except that in this case ℓ_1 maps to binary cross entropy. We set the reward function as $R = -\frac{C}{1-\text{err}}$ where err is the final classification error evaluated on the validation dataset.

Generative Adversarial Networks (GANs). Lastly, we apply AutoLoss to learn the optimization schedule of GANs. A vanilla GAN has two set of separately trained parameters: the parameters of the generator G as $\boldsymbol{\theta}_1$ and those of the discriminator D as $\boldsymbol{\theta}_2$. How to appropriately balance the optimization of $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ is a key factor that affects the success of GAN training. Beyond a manually fixed schedule, automatically adjusting the training of G and C remains untackled.

In order to express its optimization process, we let $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2\}$ ($N = 2$). $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ are trained via a minimax game

$$\min_{\boldsymbol{\theta_1}} \max_{\boldsymbol{\theta_2}} \mathcal{L}(\boldsymbol{\theta}_1, \boldsymbol{\theta}_2) = \mathbb{E}_{\boldsymbol{u}\sim p_{data}(\boldsymbol{u})}[\log D(\boldsymbol{u})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))],$$

where $\boldsymbol{z}$ is a noise variable. This is a typical alternate process that one cannot express by a linear combination of multiple loss terms (therefore cannot benefit from hyperparameter searching techniques for λ as in the previous two cases). We develop the following two terms: $\ell_1 = \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1-D(G(\boldsymbol{z})))]$, $\ell_2 = -\mathbb{E}_{\boldsymbol{u}\sim p_{data}(\boldsymbol{u})}[\log D(\boldsymbol{u})] - \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$. If we further constraint the action space of the controller to: (1) optimize ℓ_1 w.r.t. $\boldsymbol{\theta}_1$; (2) optimize ℓ_2 w.r.t. $\boldsymbol{\theta}_2$, we can exactly recover the alternate training of GANs using Eq. 1, and simplify the controller to a binary classifier.

To track the status of the training for both G and D, we mostly reuse the same four aspects of features in previous applications unless otherwise specified in the following: (1); (2) a 2D vector where the entries are the normalized L2 norms of (historical) $\nabla_{\boldsymbol{\theta}_1}\ell_1$ and $\nabla_{\boldsymbol{\theta}_2}\ell_2$, respectively, to capture how significant the updates for $\boldsymbol{\theta}_1$ and $\boldsymbol{\theta}_2$ are. In addition, we append to this vector a log ratio $\log \frac{\|\nabla_{\boldsymbol{\theta}_1}\ell_1\|_2 \cdot \sqrt{\dim(\boldsymbol{\theta}_2)}}{\|\nabla_{\boldsymbol{\theta}_2}\ell_2\|_2 \cdot \sqrt{\dim(\boldsymbol{\theta}_1)}}$, as an indicator to (hopefully) reflect how balanced their updates were; (3) A vector of most recent values of each training loss and their ratio $[\ell_1, \ell_2, \frac{\ell_1}{\ell_2}]$; (4) As there is no clearly defined metric to evaluate a GAN on a validation dataset, we come up with the following two validation metrics: for G, we generate a few samples using G with its parameters $\boldsymbol{\theta}_1^{(t)}$, and report an "inception score" (notated as $\mathcal{IS}$) on these samples as a feature to indicate how good G is. Evaluating $\mathcal{IS}$ is computationally heavy, so we only evaluate periodically and use its most recent value at t; For D, we sample a evenly distributed set of samples from both G and the training data and use the classification error of D (binary classifier for "real" or "fake") as a feature. In the same way, we instantiate the reward function as $R = C \cdot \mathcal{IS}^2$ to encourage the controller to generate schedules that lead to a better generator. Since training GAN is relatively a longer process for exploration, we set T in Eq. 3 to a fixed number (instead of the max number to convergence) to make sure there is periodically a reward signal generated to guide the controller.

[2]We recycle some notations from the description of d-ary quadratic regression to avoid clutters.

5 Evaluation

In this section, we evaluate the AutoLoss framework empirically on three task models using synthetic and real data. We reveal the follow major findings: (1) overall, AutoLoss can achieve better quality of convergence on all three tasks compared to strong baselines (section 5.1). (2) We study the generalizability of learned controller models, and show that a trained controller on a task model can be used to guide the training of another task model with different architectures, or on a totally different data distribution, while achieving faster or higher quality of convergence (section 5.2).

5.1 Quality of Convergence

We first verify the feasibility of the AutoLoss idea. We empirically show that under the formulation of Eq. 1, there *do exist* learnable update schedules, and the AutoLoss framework is able to capture its distribution and yields better quality of convergence across multiple tasks and models.

d-ary quadratic regression. We first apply AutoLoss for the d-ary quadratic regression with L1 regularization, and see whether AutoLoss can outperform its alternatives (e.g. minimizing linear combinations of loss terms). To create risks of overfitting, we generate a synthetic dataset as follows: we first sample a 16-d vector $\boldsymbol{w}$ where each entry follows uniform distribution $\mathcal{U}(-0.5, 0.5)$; we then sample P feature vectors and noise: $\boldsymbol{u}_p \sim \mathcal{U}(-5, 5)$ and $\xi_p \sim \mathcal{N}(0, 2)$, respectively, and generate dataset $\mathcal{D} = \{\boldsymbol{u}_p, v_p\}_{p=1}^{P}$ where $v_p = \boldsymbol{w}^T \cdot \boldsymbol{u} + \xi_p$. We split our dataset into five parts following [13]: $\mathcal{D}^C_{train}$ and $\mathcal{D}^C_{val}$ for training the controller; after the controller is trained, we fix its weights and use it to train a task model on the other two partitions $\mathcal{D}^T_{train}$, $\mathcal{D}^T_{val}$, in order to avoid the case that the controller just memorizes the schedule how to train the task model on $\mathcal{D}^C_{train}$. We allocate 200 samples to all four partitions, and the rest (1000 samples) for testing. In this case, the task model is highly likely to overfit the data if without proper regularizations. Our controller is a two-layer MLP with ReLU activation.

We compare `AutoLoss` to the following baselines in terms of the MSE on the test dataset in the first row of Table 1: (1) `w/o reg`: which optimizes only an MSE term on $\mathcal{D}^T_{train} \cup \mathcal{D}^T_{val}$; (2) `DGS`: we minimize the linear combination $\ell_1 + \lambda\ell_2$. To determine λ, we perform dense grid search (DGS) for λ hierarchically from coarse internals to finer ones, in total 50 experiments using data $\mathcal{D}^T_{train} \cup \mathcal{D}^T_{val}$, and report the best MSE achieved on $\mathcal{D}_{test}$ ($\lambda = 0.4$). Without regularization the performance deteriorates – we observe MSE on training set is substantially lower than that on test set. Incorporation of L1 regularization can significantly alleviate this problem, and we note `AutoLoss` is able to leverage this benefit: as long as the feature captures the states of optimization, we hypothesize AutoLoss can detect the potential risk of overfitting or needs of sparsity, and incorporate the optimization of an appropriate loss term (in time), yielding a better MSE than `w/o reg`.

Metric	`w/o reg`	`DGS`	`AutoLoss`
MSE	4.7301	4.0256	**4.0242**
err	0.124	0.0928	**0.0918**

Table 1: Comparing `AutoLoss` to `w/o` and `DGS` on d-ary regression and MLP classification tasks (averaged in 10 trials).

`AutoLoss` also achieves better results than dense `grid search`, which is a practically very strong baseline. Compared to standard SGD optimization over a linear combination of multiple loss terms, AutoLoss presents two key advantages: first, it eliminates the need to explicitly find a good λ, which sometimes is difficult and expensive, and is not transferable from one model/dataset to another. Second, AutoLoss is more flexible than optimizing a linear combination objective. Consider the case for d-ary quadratic regression where the converged solution is globally optimal and is only determined by λ. Adding a λ-scaled L1 loss term helps gain sparity, but (as a side effect) has implicitly constrained the loss surface whose global minimum might be suboptimal for the original task-specific loss. AutoLoss, on the contrast, does not restrict the loss surface to be strictly characterized in the format of a linear combination equation, therefor allows for a more flexible optimal solution that not only enjoys the positive effects brought by

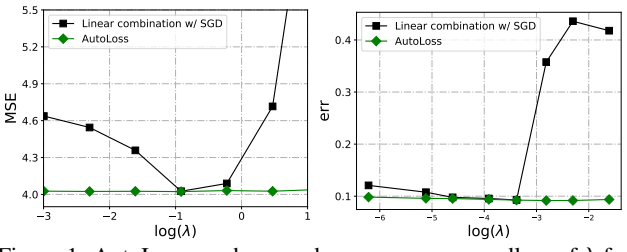


Figure 1: AutoLoss reaches good convergence regardless of λ for both d-ary regression (convex) and MLP classification (non-convex). The figures also indicate L1 is beneficial for both tasks.

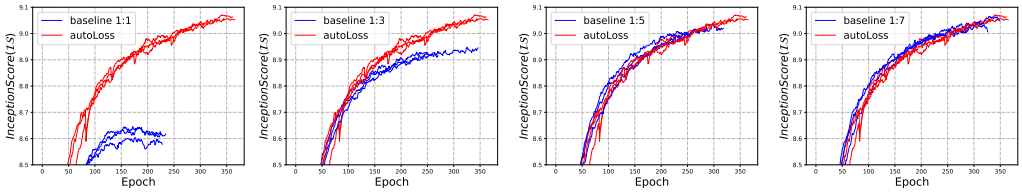


Figure 2: The training progress ($\mathcal{IR}$ vs. epochs) of four best performed baselines compared to `AutoLoss`. If an instance does not improve $\mathcal{IS}$ for 20 times, we terminate its training and regard it as converged.

regularizers, but also is much closer to the real characterization of the data distribution. To further see this, we perform an experiment where we set $\ell_2 = \lambda\|\mathbf{\Theta}\|_1$ with different λ, and find that AutoLoss is insensitive to λ (Figure 1) – it can always guide the optimization to reach the same quality of convergence (best $MSE = 4.01$) regardless of λ.

MLP classification. We apply AutoLoss to guide the training of MLP classifier as described in §4. The controller uses the same architecture as described in d-ary quadratic regression task. We create 30D synthetic data where the features are only informative in 15 dimensions (the rest of the dimensions are linear combinastions of them). We similarly compare `AutoLoss` to the two baselines `w/o reg` and `DGS` as in Table 1, and report the classification error on test dataset in the second row – `AutoLoss` again performs best against strong baselines. As shown in Figure 1, in this highly non-convex case, AutoLoss successfully learned to incorporate the L1 term to remedy over-expressiveness, and constantly guides the model to maneuver toward a better local optimal while being agnostic to the value of λ. The results suggest AutoLoss might be a better alternative to incorporate regularizations than linearly combining loss terms.

GANs. We next evaluate AutoLoss on training GANs. We employ the train split of MNIST dataset, and train GANs to generate digit images. We build the architectures of G and D exactly following [27]. As the task model itself is hard to train, in this experiment, we simply use a linear model with Bernoulli outputs as our controller. GAN loss goes beyond the format of linear combination, and there is no clear evidence or methods suggesting how the training of G and D shall be scheduled. We follow standard practice, and compare `AutoLoss` to the following baselines: (1) `GAN`: the vanilla GAN where D and G are alternately updated once a time; (2) `GAN (1:K)`: suggested by some literatures, we build a series of baselines that update D and G at the ratio $1 : K (K = 3, 5, 7, 9, 11)$ in case D is over-trained to reject all samples by G; (3) `GAN (K:1)`: a series of baselines where we contrarily bias toward more updates for the discriminator. To evaluate G, we use the inception score ($\mathcal{IS}$) [29] as a quantitative metric, and also visually inspect generated results to see if there is mode collapse. To calculate $\mathcal{IS}$ of digit images, we follow [11], and train a CNN classifier on the MNIST training split and use it as the "inception network" to report $\mathcal{IS}$ (the network has $\mathcal{IR} = 9.5$ on groundtruth MNIST images). In Figure 2, we plot the $\mathcal{IS}$[3] w.r.t. number of training epochs, comparing `AutoLoss` to `GAN` and another three best performed baselines from `GAN (1:K)` and `GAN (K:1)`, each with three trials of experiments. We also report the converged $\mathcal{IS}$ for all comparators here: 8.6307, 8.9353, 9.0206, 9.0415, **9.0549** for `GAN`, `GAN (1:3)`, `GAN (1:5)`, `GAN (1:7)`, `AutoLoss`, respectively.

In general, GANs trained with AutoLoss present two improvements over baselines: (1) it achieves higher quality of final convergence in terms of inception score; (2) it has faster per-epoch convergence. For example, when comparing to a normal `GAN`, `AutoLoss` improves the converged $\mathcal{IR}$ for 0.5, and is almost 3x faster to achieve where `GAN` converges ($\mathcal{IS} = 8.6$) in average. We empirically observe the `GAN (1:7)` performs closest to `AutoLoss` and is the best fixed update schedule we have ever found: it achieves the best $\mathcal{IS}$ 9.0415, compared to `AutoLoss` 9.0549, though almost 5 epochs slower in average. This echoes the statement from [2, 17, 11] that more updates of G over D might be preferable to GAN training. It is worth mentioning that all `GAN K:1` baselines perform worse than the rest and are skipped in Figure 2. We visualize some generated digit images by AutoLoss-guided GAN in the left of Figure 3.

5.2 Transferability

We next investigate the generalization ability of a trained controller on different models or datasets.

[3]While we are aware of that inception score has recently been doubted [4] as a proper metric to evaluate generators, we visualize the generated samples and observe in our experiments that it is directly relevant with the quality of generated images, therefore could be used as an indicator of the convergence

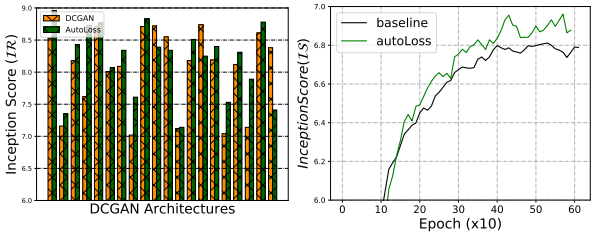
Figure 3: Images generated by AutoLoss-guided GAN. The meta model is trained on MNIST dataset and applied to guide the training of GAN on MNIST (left) and CIFAR10 (right).

Transfer to different models. To see whether a differently configured task model can benefit from a trained controller (i.e. the controller is not memorizing the optimization behavior of the specific model it is trained with), we design the following experiment: we first train a GAN on MNIST; we then change the neural network architecture of the GAN, and let the controller guide the training of the new GAN $(1:1)$ from scratch, on the same dataset. We compare the averaged converged $\mathcal{IR}$ of the trained model between with and without the AutoLoss controller, as in Figure 4(b). Clearly, we see `AutoLoss` outperforms `DCGAN` in 16 out of 20 architectures. As we only report architectures that can successfully learn meaningful digit generators (with converged $\mathcal{IS} > 6$), we can conclude the controller is able to help an unseen architecture achieve a better quality of convergence (for failed cases, the results are meaningless).

Transfer to different data distributions. Our second set of experiments try to figure out whether an AutoLoss controller is generally applicable across different data distributions. Specifically, we first train a controller for a task model on one dataset. We then fix the parameters of the controller, and use it to guide the training of the same task model from scratch, but on other dataset with totally different distributions. We compare the training of the AutoLoss-guided model with the vanilla one. We report the comparison results in Table 4(a) and Figure 4(b) one two tasks: (1) MLP classifier, that we train the controller using a dataset generated following a process. We then generate another three sets of synthetic samples using different specifications of the process (therefore changes of distributions); (b) GANs, where we first train a controller for digit generation on MNIST, and use the controller to guide the training of the same GAN architecture on CIFAR-10. In both cases we observe AutoLoss managed to use it knowledge to guide the model training on unseen data. On MLP classification, it delivers trained models comparable to or better than models searched via `DGS`; On image generation, when transferred from digit images to natural images, a controller guided GAN achieves both higher quality of convergence, and per-epoch convergence, than a normal GAN trained with $1-1$ schedule, as shown in the right picture in Figure 3 and Figure 4(c).

Dataset #	`w/o reg`	`DGS`	`AutoLoss`
1	0.1337	**0.1019**	0.1037
2	0.1294	0.1035	**0.1016**
3	0.1318	0.1022	**0.0997**

(a)



(b) (c)

Figure 4: (a) Comparing the MLP classification results for the data transfer experiments. (b) Comparing the final convergence (in terms of $\mathcal{IS}$) of randomly sampled DCGAN architectures trained with and without a trained AutoLoss controller. We randomly sampled DCGAN architectures [27, 29] by sampling: (1) # of filters in the base layer of G and D from $\{32, 64, 128\}$; (2) dim($\boldsymbol{z}$) from $\{64, 128\}$; (3) whether to use batchnorm or not; (4) the activation functions from $\{\text{ReLU}, \text{LeakyReLU}\}$. (c) The training progress ($\mathcal{IR} vs. epochs$ comparison of a vanilla GAN and a AutoLoss-guided GAN on CIFAR-10.)

6 Conclusion

This paper proposed a unified formulation for iterative and alternate optimization processes and developed AutoLoss, a framework to automatically generate an optimization schedule which can helps achieve better quality of convergence. Comprehensive experiments on synthetic and real data have demonstrated that the optimization schedule produced by AutoLoss controller can guide the task model to find a better optima and also has a good transferability when applied to different models or different data distributions.

References

[1] 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. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[3] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

[4] Shane Barratt and Rishi Sharma. A note on the inception score. *arXiv preprint arXiv:1801.01973*, 2018.

[5] Irwan Bello, Barret Zoph, Vijay Vasudevan, and Quoc V Le. Neural optimizer search with reinforcement learning. *arXiv preprint arXiv:1709.07417*, 2017.

[6] David M Blei, Alp Kucukelbir, and Jon D McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

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

[8] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[9] Stephen Boyd, Lin Xiao, and Almir Mutapcic. Subgradient methods. 2003.

[10] Yutian Chen, Matthew W Hoffman, Sergio Gómez Colmenarejo, Misha Denil, Timothy P Lillicrap, Matt Botvinick, and Nando de Freitas. Learning to learn without gradient descent by gradient descent. *arXiv preprint arXiv:1611.03824*, 2016.

[11] Zhijie Deng, Hao Zhang, Xiaodan Liang, Luona Yang, Shizhen Xu, Jun Zhu, and Eric P Xing. Structured generative adversarial networks. In *Advances in Neural Information Processing Systems*, pages 3902–3912, 2017.

[12] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[13] Yang Fan, Fei Tian, Tao Qin, Xiang-Yang Li Li, and Tie-Yan Liu. Learning to teach. *arXiv preprint arXiv:1606.01885*, 2018.

[14] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

[15] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[16] Thomas L Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National academy of Sciences*, 101(suppl 1):5228–5235, 2004.

[17] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. *arXiv preprint arXiv:1704.00028*, 2017.

[18] Matthew D Hoffman, David M Blei, Chong Wang, and John Paisley. Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347, 2013.

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

[20] Ke Li and Jitendra Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.

[21] Yi-An Ma, Tianqi Chen, and Emily Fox. A complete recipe for stochastic gradient mcmc. In *Advances in Neural Information Processing Systems*, pages 2917–2925, 2015.

[22] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International Conference on Machine Learning*, pages 2113–2122, 2015.

[23] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

[24] Todd K Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60, 1996.

[25] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural networks*, 21(4):682–697, 2008.

[26] Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*, 2018.

[27] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[28] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. 2016.

[29] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, pages 2226–2234, 2016.

[30] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[31] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

[32] Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

[33] Max Welling and Yee W Teh. Bayesian learning via stochastic gradient langevin dynamics. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 681–688, 2011.

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

[35] Stephen J Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

[36] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.