LEARNING TO OPTIMIZE

Ke Li & Jitendra Malik
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720
United States
{ke.li,malik}@eecs.berkeley.edu

ABSTRACT

Algorithm design is a laborious process and often requires many iterations of ideation and validation. In this paper, we explore automating algorithm design and present a method to *learn* an optimization algorithm. We approach this problem from a reinforcement learning perspective and represent any particular optimization algorithm as a policy. We learn an optimization algorithm using guided policy search and demonstrate that the resulting algorithm outperforms existing hand-engineered algorithms in terms of convergence speed and/or the final objective value.

1 INTRODUCTION

Continuous optimization algorithms are some of the most ubiquitous tools used in virtually all areas of science and engineering. Indeed, they are the workhorse of machine learning and power most learning algorithms. Consequently, optimization difficulties become learning challenges – because their causes are often not well understood, they are one of the most vexing issues that arise in practice. One solution is to design better optimization algorithms that are immune to these failure cases. This requires careful analysis of existing optimization algorithms and clever solutions to overcome their weaknesses; thus, doing so is both laborious and time-consuming. Is there a better way? If the mantra of machine learning is to learn what is traditionally manually designed, why not take it a step further and *learn* the optimization algorithm itself?

Consider the general structure of an algorithm for unconstrained continuous optimization, which is outlined in Algorithm 1. Starting from a random location in the domain of the objective function, the algorithm iteratively updates the current iterate by a step vector Δx computed from some functional π of the objective function, the current iterate and past iterates.

Algorithm 1 General structure of unconstrained optimization algorithms

Require: Objective function f
  $x^{(0)} \leftarrow$ random point in the domain of f
  for $i = 1, 2, \ldots$ **do**
    $\Delta x \leftarrow \pi(f, \{x^{(0)}, \ldots, x^{(i-1)}\})$
    if stopping condition is met **then**
      return $x^{(i-1)}$
    end if
    $x^{(i)} \leftarrow x^{(i-1)} + \Delta x$
  end for

Different optimization algorithms only differ in the choice of the update formula π. Examples of existing optimization algorithms and their corresponding update formulas are shown in Table 1.

If we can learn π, we will be able to learn an optimization algorithm. Since it is difficult to model general functionals, in practice, we restrict the dependence of π on the objective function f to objective values and gradients evaluated at current and past iterates. Hence, π can be simply modelled as a function from the objective values and gradients along the trajectory taken by the optimizer so far

Algorithm	Update Formula π
Gradient Descent	$\pi(\cdot) = -\gamma \nabla f(x^{(i-1)})$
Momentum	$\pi(\cdot) = -\gamma \left(\sum_{j=0}^{i-1} \alpha^{i-1-j} \nabla f(x^{(j)}) \right)$
Conjugate Gradient	$\pi(\cdot) = -\gamma \left(\nabla f(x^{(i-1)}) + \sum_{j=0}^{i-2} \left(\frac{\left\|\nabla f(x^{(j+1)})\right\|^2}{\left\|\nabla f(x^{(j)})\right\|^2} \right)^{i-1-j} \nabla f(x^{(j)}) \right)$

Table 1: Choices of the update formula π made by hand-engineered optimization algorithms. We propose learning π automatically in the hope of learning an optimization algorithm that converges faster and to better optima on objective functions of interest.

to the next step vector. If we model π with a universal function approximator like a neural net, it is then possible to search over the space of optimization algorithms by learning the parameters of the neural net. We formulate this as a reinforcement learning problem, where any particular optimization algorithm simply corresponds to a policy. Learning an optimization algorithm then reduces to finding an optimal policy. For this purpose, we use an off-the-shelf reinforcement learning algorithm known as guided policy search (Levine & Abbeel, 2014), which has demonstrated success in a variety of robotic control settings (Levine et al., 2015a; Finn et al., 2015; Levine et al., 2015b; Han et al., 2015).

Our goal is to learn about regularities in the geometry of the error surface induced by a class of objective functions of interest and exploit this knowledge to optimize the class of objective functions faster. This is potentially advantageous, since the learned optimizer is trained on the actual objective functions that arise in practice, whereas hand-engineered optimizers are often analyzed in the convex setting and applied to the non-convex setting.

1.1 Learning How to Learn

When the objective functions under consideration correspond to loss functions for training a model, the proposed framework effectively learns how to learn. The loss function for training a model on a particular task/dataset is a particular objective function, and so the loss on many tasks corresponds to a set of objective functions. We can train an optimizer on this set of objective functions, which can hopefully learn to exploit regularities of the model and train it faster irrespective of the task. As a concrete example, if the model is a neural net with ReLU activations, our goal is to learn an optimizer that can leverage the piecewise linearity of the model.

We evaluate the learned optimizer on its ability to generalize to unseen objective functions. Akin to the supervised learning paradigm, we divide the dataset of objective functions into training and test sets. At test time, the learned optimizer can be used stand-alone and functions exactly like a hand-engineered optimizer, except that the update formula is replaced with a neural net and no hyperparameters like step size or momentum need to be specified by the user. In particular, it does not perform multiple trials on the same objective function at test time, unlike hyperparameter optimization. Since different objective functions correspond to the loss for training a model on different tasks, the optimizer is effectively asked to train on its experience of learning on some tasks and generalize to other possibly unrelated tasks. It is therefore critical to ensure that the optimizer does *not* learn anything about particular tasks; this would be considered as overfitting under this setting. Notably, this goal is different from the line of work on "learning to learn" or "meta-learning", whose goal is to learn something about a family of tasks. To prevent overfitting to particular tasks, we train the optimizer to learn on randomly generated tasks.

2 Related Work

2.1 Meta-Learning

When the objective functions optimized by the learned optimizer correspond to loss functions for training a model, learning the optimizer can be viewed as learning how to learn. This theme of learning about learning itself has been explored and is referred to as "learning to learn" or "meta-

learning" (Baxter et al., 1995; Vilalta & Drissi, 2002; Brazdil et al., 2008; Thrun & Pratt, 2012). Various authors have used the term in different ways and there is no consensus on its precise definition. While there is agreement on what kinds of knowledge should be learned at the base-level, it is less clear what kinds of meta-knowledge should be learned at the meta-level. We briefly summarize the various perspectives that have been presented below.

One form of meta-knowledge is the commonalities across a family of related tasks (Abu-Mostafa, 1993). Under this framework, the goal of the meta-learner is to learn about such commonalities, so that given the learned commonalities, base-level learning on new tasks from the family can be performed faster. This line of work is often better known as transfer learning and multi-task learning.

A different approach (Brazdil et al., 2003) is to learn how to select the base-level learner that achieves the best performance for a given task. Under this setting, the meta-knowledge is the correlation between properties of tasks and the performance of different base-level learners trained on them. There are two challenges associated with this approach: the need to devise meta-features on tasks that capture similarity between different tasks, and the need to parameterize the space of base-level learners to make search in this space tractable.

Schmidhuber (2004) proposes representing base-level learners as general-purpose programs, that is, sequences of primitive operations. While such a representation can in principle encode all base-level learners, searching in this space takes exponential time in the length of the target program.

The proposed method differs from these lines in work in several important ways. First, the proposed method learns regularities in the optimization/learning process itself, rather than regularities that are shared by different tasks or regularities in the mapping between tasks and best-performing base-level learners. More concretely, the meta-knowledge in the proposed framework can capture regularities in the error surface. Second, unlike the approaches above, we explicitly aim to avoid capturing any regularities about the task. Under the proposed framework, only model-specific regularities are captured at the meta-level, while task-specific regularities are captured at the base-level.

2.2 Program Induction

Because the proposed method learns an algorithm, it is related to program induction, which considers the problem of learning programs from examples of input and output. Several different approaches have been proposed: genetic programming (Cramer, 1985) represents programs as abstract syntax trees and evolves them using genetic algorithms, Liang et al. (2010) represents programs explicitly using a formal language, constructs a hierarchical Bayesian prior over programs and performs inference using an MCMC sampling procedure and Graves et al. (2014) represents programs implicitly as sequences of memory access operations and trains a recurrent neural net to learn the underlying patterns in the memory access operations. Hochreiter et al. (2001) considers the special case of online learning algorithms, each of which is represented as a recurrent neural net with a particular setting of weights, and learns the online learning algorithm by learning the neural net weights. While the program/algorithm improves as training progresses, the algorithms learned using these methods have not been able to match the performance of simple hand-engineered algorithms. In contrast, our aim is learn an algorithm that is better than known hand-engineered algorithms.

2.3 Hyperparameter Optimization

There is a large body of work on hyperparameter optimization, which studies the optimization of hyperparameters used to train a model, such as the learning rate, the momentum decay factor and regularization parameters. Most methods (Hutter et al., 2011; Bergstra et al., 2011; Snoek et al., 2012; Swersky et al., 2013; Feurer et al., 2015) rely on sequential model-based Bayesian optimization (Mockus et al., 1978; Brochu et al., 2010), while others adopt a random search approach (Bergstra & Bengio, 2012) or use gradient-based optimization (Bengio, 2000; Domke, 2012; Maclaurin et al., 2015). Because each hyperparameter setting corresponds to a particular instantiation of an optimization algorithm, these methods can be viewed as a way to search over different instantiations of the same optimization algorithm. The proposed method, on the other hand, can search over a larger space of possible optimization algorithms. In addition, as noted previously, when presented with a new objective function at test time, the learned optimizer does not need to conduct multiple trials with different hyperparameter settings.

2.4 SPECIAL CASES AND OTHER RELATED WORK

Work on online hyperparameter adaptation studies ways to choose the step size or other hyperparameters adaptively while performing optimization. Stochastic meta-descent (Bray et al., 2004) derives a rule for adaptively choosing the step size, Ruvolo et al. (2009) learns a policy for picking the damping factor in the Levenberg-Marquardt algorithm and recent work (Hansen, 2016; Daniel et al., 2016; Fu et al., 2016) explores learning a policy for choosing the step size. Unlike this line of work, the proposed method learns a policy for choosing the step direction as well as step size, thereby making it possible to learn a new optimization algorithm that is different from known algorithms. A different line of work (Gregor & LeCun, 2010; Sprechmann et al., 2013) explores learning special-purpose solvers for a class of optimization problems that arise in sparse coding.

Work that appeared on ArXiv after this paper (Andrychowicz et al., 2016) explores a similar theme under a different setting, where the goal is learn a task-dependent optimization algorithm. The optimizer is trained from the experience of training on a particular task or family of tasks and is evaluated on its ability to train on the same task or family of tasks. Under this setting, the optimizer learns regularities about the task itself rather than regularities of the model in general.

3 BACKGROUND ON REINFORCEMENT LEARNING

3.1 MARKOV DECISION PROCESS

In the reinforcement learning setting, the learner is given a choice of actions to take in each time step, which changes the state of the environment in an unknown fashion, and receives feedback based on the consequence of the action. The feedback is typically given in the form of a reward or cost, and the objective of the learner is to choose a sequence of actions based on observations of the current environment that maximizes cumulative reward or minimizes cumulative cost over all time steps.

More formally, a reinforcement learning problem can be characterized by a Markov decision process (MDP). We consider an undiscounted finite-horizon MDP with continuous state and action spaces defined by the tuple $(\mathcal{S}, \mathcal{A}, p_0, p, c, T)$, where $\mathcal{S} \subseteq \mathbb{R}^D$ is the set of states, $\mathcal{A} \subseteq \mathbb{R}^d$ is the set of actions, $p_0 : \mathcal{S} \to \mathbb{R}^+$ is the probability density over initial states, $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}^+$ is the transition probability density, that is, the conditional probability density over successor states given the current state and action, $c : \mathcal{S} \to \mathbb{R}$ is a function that maps state to cost and T is the time horizon. A policy $\pi : \mathcal{S} \times \mathcal{A} \times \{0, \ldots, T-1\} \to \mathbb{R}^+$ is a conditional probability density over actions given the state at each time step. When a policy is independent of the time step, it is referred to as stationary.

3.2 POLICY SEARCH

This problem of finding the cost-minimizing policy is known as the policy search problem. More precisely, the objective is to find a policy π^* such that

$$\pi^* = \arg\min_{\pi} \mathbb{E}_{s_0, a_0, s_1, \ldots, s_T} \left[\sum_{t=0}^{T} c(s_t) \right],$$

where the expectation is taken with respect to the joint distribution over the sequence of states and actions, often referred to as a trajectory, which has the density

$$q(s_0, a_0, s_1, \ldots, s_T) = p_0(s_0) \prod_{t=0}^{T-1} \pi(a_t | s_t, t)\, p(s_{t+1} | s_t, a_t).$$

To enable generalization to unseen states, the policy is typically parameterized and minimization is performed over representable policies. Solving this problem exactly is intractable in all but selected special cases. Therefore, policy search methods generally tackle this problem by solving it approximately. In addition, the transition probability density p is typically not known, but may be accessed via sampling.

3.3 GUIDED POLICY SEARCH

Guided policy search (GPS) (Levine & Abbeel, 2014) is a method for searching over expressive non-linear policy classes in continuous state and action spaces. It works by alternating between computing a mixture of target trajectories and training the policy to replicate them. Successive iterations locally improve target trajectories while ensuring proximity to behaviours that are reproducible by the policy. Target trajectories are computed by fitting local approximations to the cost and transition probability density and optimizing over a restricted class of time-varying linear target policies subject to a trust region constraint. The stationary non-linear policy is trained to minimize the squared Mahalanobis distance between the predicted and target actions at each time step.

More precisely, GPS works by solving the following constrained optimization problem:

$$\min_{\theta,\eta} \mathbb{E}_{\psi}\left[\sum_{t=0}^{T} c(s_t)\right] \text{ s.t. } \psi\left(a_t \middle| s_t, t; \eta\right) = \pi\left(a_t \middle| s_t; \theta\right) \ \forall a_t, s_t, t,$$

where ψ denotes the time-varying target policy, π denotes the stationary non-linear policy, and $\mathbb{E}_{\psi}[\cdot]$ denotes the expectation taken with respect to the trajectory induced by the target policy ψ. ψ is assumed to be conditionally Gaussian whose mean is linear in s_t and π is assumed to be conditionally Gaussian whose mean could be an arbitrary function of s_t. To solve this problem, the equality constraint is relaxed and replaced with a penalty on the KL-divergence between ψ and π. Different flavours of GPS (Levine & Abbeel, 2014; Levine et al., 2015a) use different constrained optimization methods, which all involve alternating between optimizing the parameters of ψ and π.

For updating ψ, GPS first builds a model $\tilde{p}$ of the transition probability density p of the form $\tilde{p}\left(s_{t+1} \middle| s_t, a_t, t\right) \coloneqq \mathcal{N}(A_t s_t + B_t a_t + c_t, F_t)$[1], where A_t, B_t and c_t are parameters estimated from samples drawn from the trajectory induced by the existing ψ. It also computes local quadratic approximations to the cost, so that $c(s_t) \approx \frac{1}{2} s_t^T C_t s_t + d_t^T s_t + h_t$ for s_t's that are near the samples. It then solves the following:

$$\min_{K_t, k_t, G_t} \mathbb{E}_{\tilde{\psi}}\left[\sum_{t=0}^{T} \frac{1}{2} s_t^T C_t s_t + d_t^T s_t\right]$$
$$\text{s.t.} \sum_{t=0}^{T} D_{KL}\left(p\left(s_t\right)\psi\left(\cdot \middle| s_t, t; \eta\right) \middle\| p\left(s_t\right)\psi\left(\cdot \middle| s_t, t; \eta'\right)\right) \leq \epsilon,$$

where $\mathbb{E}_{\tilde{\psi}}[\cdot]$ denotes the expectation taken with respect to the trajectory induced by the target policy ψ if states transition according to the model $\tilde{p}$. K_t, k_t, G_t are the parameters of $\psi\left(a_t \middle| s_t, t; \eta\right) \coloneqq \mathcal{N}\left(K_t s_t + k_t, G_t\right)$ and η' denotes the parameters of the previous target policy. It turns out that this optimization problem can be solved in closed form using a dynamic programming algorithm known as linear-quadratic-Gaussian regulator (LQG).

For updating π, GPS minimizes $D_{KL}\left(p\left(s_t\right)\pi\left(\cdot \middle| s_t\right) \middle\| p\left(s_t\right)\psi\left(\cdot \middle| s_t, t\right)\right)$. Assuming fixed covariance and omitting dual variables, this corresponds to minimizing the following:

$$\mathbb{E}_{\psi}\left[\sum_{t=0}^{T} \left(\mathbb{E}_{\pi}\left[a_t \middle| s_t\right] - \mathbb{E}_{\psi}\left[a_t \middle| s_t, t\right]\right)^T G_t^{-1} \left(\mathbb{E}_{\pi}\left[a_t \middle| s_t\right] - \mathbb{E}_{\psi}\left[a_t \middle| s_t, t\right]\right)\right],$$

where $\mathbb{E}_{\pi}[\cdot]$ denotes the expectation taken with respect to the trajectory induced by the non-linear policy π. We refer interested readers to (Levine & Abbeel, 2014) and (Levine et al., 2015a) for details.

4 FORMULATION

We observe that the execution of an optimization algorithm can be viewed as the execution of a particular policy in an MDP: the state consists of the current iterate and the objective values and gradients evaluated at the current and past iterates, the action is the step vector that is used to update

[1] In a slight abuse of notation, we use $\mathcal{N}(\mu, \Sigma)$ to denote the density of a Gaussian distribution with mean μ and covariance Σ.

the current iterate, and the transition probability is partially characterized by the update formula, $x^{(i)} \leftarrow x^{(i-1)} + \Delta x$. The policy that is executed corresponds precisely to the choice of π used by the optimization algorithm. For this reason, we will also use π to denote the policy at hand. Under this formulation, searching over policies corresponds to searching over possible optimization algorithms.

To learn π, we need to define the cost function, which should penalize policies that exhibit undesirable behaviours during their execution. Since the performance metric of interest for optimization algorithms is the speed of convergence, the cost function should penalize policies that converge slowly. To this end, assuming the goal is to minimize the objective function, we define cost at a state to be the objective value at the current iterate. This encourages the policy to reach the minimum of the objective function as quickly as possible. We choose to parameterize the mean of π using a neural net, due to its appealing properties as a universal function approximator and strong empirical performance in a variety of applications. We use GPS to learn π.

5 Implementation Details

We store the current iterate, previous gradients and improvements in the objective value from previous iterations in the state. We keep track of only the information pertaining to the previous H time steps and use $H = 25$ in our experiments. More specifically, the dimensions of the state space encode the following information:

- Current iterate
- Change in the objective value at the current iterate relative to the objective value at the i^{th} most recent iterate for all $i \in \{2, \ldots, H+1\}$
- Gradient of the objective function evaluated at the i^{th} most recent iterate for all $i \in \{2, \ldots, H+1\}$

Initially, we set the dimensions corresponding to historical information to zero. The current iterate is only used to compute the cost; because the policy should not depend on the absolute coordinates of the current iterate, we exclude it from the input that is fed into the neural net.

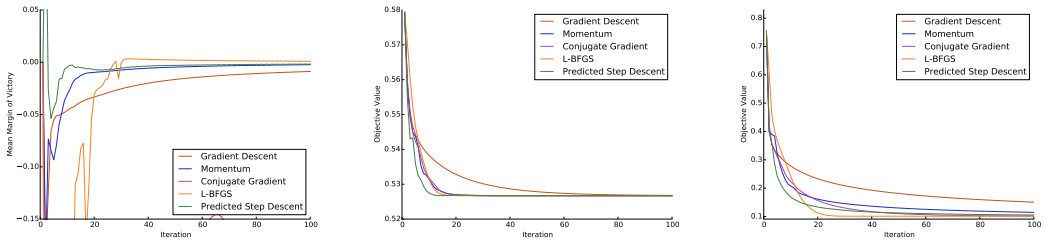
We use a small neural net with a single hidden layer of 50 hidden units to model the mean of π. Softplus activation units are used at the hidden layer and linear activation units are used at the output layer. We initialize the weights of the neural net randomly and do not regularize the magnitude of weights.

Initially, we set the target trajectory distribution so that the mean action given state at each time step matches the step vector used by the gradient descent method with momentum. We choose the best settings of the step size and momentum decay factor for each objective function in the training set by performing a grid search over hyperparameters and running noiseless gradient descent with momentum for each hyperparameter setting. We use a mixture of 10 Gaussians as a prior for fitting the parameters of the transition probability density.

For training, we sample 20 trajectories with a length of 40 time steps for each objective function in the training set. After each iteration of guided policy search, we sample new trajectories from the new distribution and discard the trajectories from the preceding iteration.

6 Experiments

We learn optimization algorithms for various convex and non-convex classes of objective functions that correspond to loss functions for different machine learning models. We learn an optimizer for logistic regression, robust linear regression using the Geman-McClure M-estimator and a two-layer neural net classifier with ReLU activation units. The geometry of the error surface becomes progressively more complex: the loss for logistic regression is convex, the loss for robust linear regression is non-convex, and the loss for the neural net has many local minima.



(a) (b) (c)

Figure 1: (a) Mean margin of victory of each algorithm for optimizing the logistic regression loss. Higher margin of victory indicates better performance. (b-c) Objective values achieved by each algorithm on two objective functions from the test set. Lower objective values indicate better performance. Best viewed in colour.

6.1 Logistic Regression

We consider a logistic regression model with an ℓ_2 regularizer on the weight vector. Training the model requires optimizing the following objective:

$$\min_{\mathbf{w},b} -\frac{1}{n}\sum_{i=1}^{n} y_i \log \sigma\left(\mathbf{w}^T\mathbf{x}_i + b\right) + (1-y_i)\log\left(1-\sigma\left(\mathbf{w}^T\mathbf{x}_i+b\right)\right) + \frac{\lambda}{2}\left\|\mathbf{w}\right\|_2^2,$$

where $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ denote the weight vector and bias respectively, $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{0, 1\}$ denote the feature vector and label of the i^{th} instance, λ denotes the coefficient on the regularizer and $\sigma(z) := \frac{1}{1+e^{-z}}$. For our experiments, we choose $\lambda = 0.0005$ and $d = 3$. This objective is convex in $\mathbf{w}$ and b.

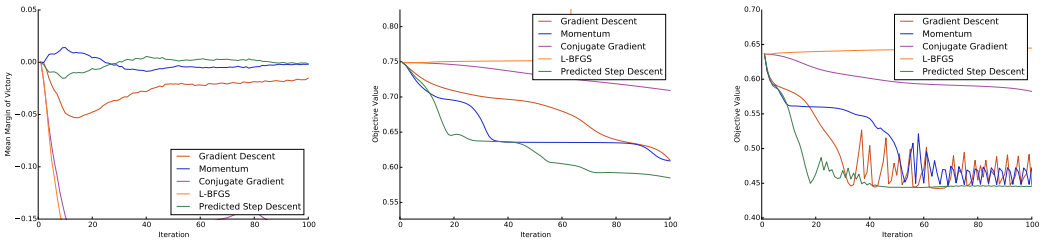
We train an algorithm for optimizing objectives of this form. Different examples in the training set correspond to such objective functions with different instantiations of the free variables, which in this case are $\mathbf{x}_i$ and y_i. Hence, each objective function in the training set corresponds to a logistic regression problem on a different dataset.

To construct the training set, we randomly generate a dataset of 100 instances for each function in the training set. The instances are drawn randomly from two multivariate Gaussians with random means and covariances, with half drawn from each. Instances from the same Gaussian are assigned the same label and instances from different Gaussians are assigned different labels.

We train the optimizer on a set of 90 objective functions. We evaluate it on a test set of 100 random objective functions generated using the same procedure and compare to popular hand-engineered algorithms, such as gradient descent, momentum, conjugate gradient and L-BFGS. All baselines are run with the best hyperparameter settings tuned on the training set.

For each algorithm and objective function in the test set, we compute the difference between the objective value achieved by a given algorithm and that achieved by the best of the competing algorithms at every iteration, a quantity we will refer to as "the margin of victory". This quantity is positive when the current algorithm is better than all other algorithms and negative otherwise. In Figure 1a, we plot the mean margin of victory of each algorithm at each iteration averaged over all objective functions in the test set.

As shown, the learned optimizer, which we will henceforth refer to as "predicted step descent", outperforms gradient descent, momentum and conjugate gradient at almost every iteration. The margin of victory for predicted step descent is high in early iterations, indicating that it converges much faster than other algorithms. It is interesting to note that despite having seen only trajectories of length 40 at training time, the learned optimizer is able to generalize to much longer time horizons at test time. L-BFGS converges to slightly better optima than predicted step descent and the momentum method. This is not surprising, as the objective functions are convex and L-BFGS is known to be a very good optimizer for convex problems.



(a) (b) (c)

Figure 2: (a) Mean margin of victory of each algorithm for optimizing the robust linear regression loss. Higher margin of victory indicates better performance. (b-c) Objective values achieved by each algorithm on two objective functions from the test set. Lower objective values indicate better performance. Best viewed in colour.

We show the performance of each algorithm on two objective functions from the test set in Figures 1b and 1c. In Figure 1b, predicted step descent converges faster than all other algorithms. In Figure 1c, predicted step descent initially converges faster than all other algorithms but is later overtaken by L-BFGS, while remaining faster than all other optimizers. However, it eventually achieves the same objective value as L-BFGS, while the objective values achieved by gradient descent and momentum remain much higher.

6.2 Robust Linear Regression

Next, we consider the problem of linear regression using a robust loss function. One way to ensure robustness is to use an M-estimator for parameter estimation. A popular choice is the Geman-McClure estimator, which induces the following objective:

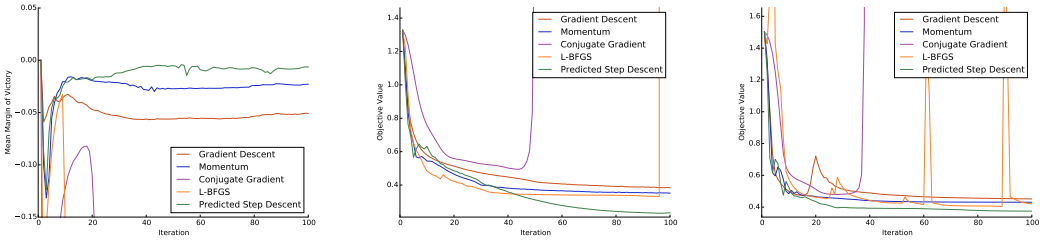
$$\min_{\mathbf{w},b} \frac{1}{n} \sum_{i=1}^{n} \frac{\left(y_i - \mathbf{w}^T \mathbf{x}_i - b\right)^2}{c^2 + \left(y_i - \mathbf{w}^T \mathbf{x}_i - b\right)^2},$$

where $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ denote the weight vector and bias respectively, $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$ denote the feature vector and label of the i^{th} instance and $c \in \mathbb{R}$ is a constant that modulates the shape of the loss function. For our experiments, we use $c = 1$ and $d = 3$. This loss function is not convex in either $\mathbf{w}$ or b.

As with the preceding section, each objective function in the training set is a function of the above form with a particular instantiation of $\mathbf{x}_i$ and y_i. The dataset for each objective function is generated by drawing 25 random samples from each one of four multivariate Gaussians, each of which has a random mean and the identity covariance matrix. For all points drawn from the same Gaussian, their labels are generated by projecting them along the same random vector, adding the same randomly generated bias and perturbing them with i.i.d. Gaussian noise.

The optimizer is trained on a set of 120 objective functions. We evaluate it on 100 randomly generated objective functions using the same metric as above. As shown in Figure 2a, predicted step descent outperforms all hand-engineered algorithms except at early iterations. While it dominates gradient descent, conjugate gradient and L-BFGS at all times, it does not make progress as quickly as the momentum method initially. However, after around 30 iterations, it is able to close the gap and surpass the momentum method. On this optimization problem, both conjugate gradient and L-BFGS diverge quickly. Interestingly, unlike in the previous experiment, L-BFGS no longer performs well, which could be caused by non-convexity of the objective functions.

Figures 2b and 2c show performance on objective functions from the test set. In Figure 2b, predicted step descent not only converges the fastest, but also reaches a better optimum than all other algorithms. In Figure 2c, predicted step descent converges the fastest and is able to avoid most of the oscillations that hamper gradient descent and momentum after reaching the optimum.



(a) (b) (c)

Figure 3: (a) Mean margin of victory of each algorithm for training neural net classifiers. Higher margin of victory indicates better performance. (b-c) Objective values achieved by each algorithm on two objective functions from the test set. Lower objective values indicate better performance. Best viewed in colour.

6.3 Neural Net Classifier

Finally, we train an optimizer to train a small neural net classifier. We consider a two-layer neural net with ReLU activation on the hidden units and softmax activation on the output units. We use the cross-entropy loss combined with ℓ_2 regularization on the weights. To train the model, we need to optimize the following objective:

$$\min_{W,U,\mathbf{b},\mathbf{c}} -\frac{1}{n}\sum_{i=1}^{n}\log\left(\frac{\exp\left((U\max\left(W\mathbf{x}_i+\mathbf{b},0\right)+\mathbf{c})_{y_i}\right)}{\sum_j \exp\left((U\max\left(W\mathbf{x}_i+\mathbf{b},0\right)+\mathbf{c})_j\right)}\right)+\frac{\lambda}{2}\left\|W\right\|_F^2+\frac{\lambda}{2}\left\|U\right\|_F^2,$$

where $W \in \mathbb{R}^{h\times d}, b \in \mathbb{R}^h, U \in \mathbb{R}^{p\times h}, c \in \mathbb{R}^p$ denote the first-layer and second-layer weights and biases, $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \{1,\ldots,p\}$ denote the input and target class label of the i^{th} instance, λ denotes the coefficient on regularizers and $(\mathbf{v})_j$ denotes the j^{th} component of $\mathbf{v}$. For our experiments, we use $\lambda = 0.0005$ and $d = h = p = 2$. The error surface is known to have complex geometry and multiple local optima, making this a challenging optimization problem.

The training set consists of 80 objective functions, each of which corresponds to the objective for training a neural net on a different dataset. Each dataset is generated by generating four multivariate Gaussians with random means and covariances and sampling 25 points from each. The points from the same Gaussian are assigned the same random label of either 0 or 1. We make sure not all of the points in the dataset are assigned the same label.

We evaluate the learned optimizer in the same manner as above. As shown in Figure 3a, predicted step descent significantly outperforms all other algorithms. In particular, as evidenced by the sizeable and sustained gap between margin of victory for predicted step descent and the momentum method, predicted step descent is able to reach much better optima and is less prone to getting trapped in local optima compared to other methods. This gap is also larger compared to that exhibited in previous sections, suggesting that hand-engineered algorithms are more sub-optimal on challenging optimization problems and so the potential for improvement from learning the algorithm is greater in such settings. Due to non-convexity, conjugate gradient and L-BFGS often diverge.

Performance on examples of objective functions from the test set is shown in Figures 3b and 3c. As shown, predicted step descent is able to reach better optima than all other methods and largely avoids oscillations that other methods suffer from.

6.4 Visualization of Optimization Trajectories

We visualize optimization trajectories followed by the learned algorithm and various hand-engineered algorithms to gain further insights into the behaviour of the learned algorithm. We generated random two-dimensional logistic regression problems and plot trajectories followed by different algorithms on each problem in Figure 4.

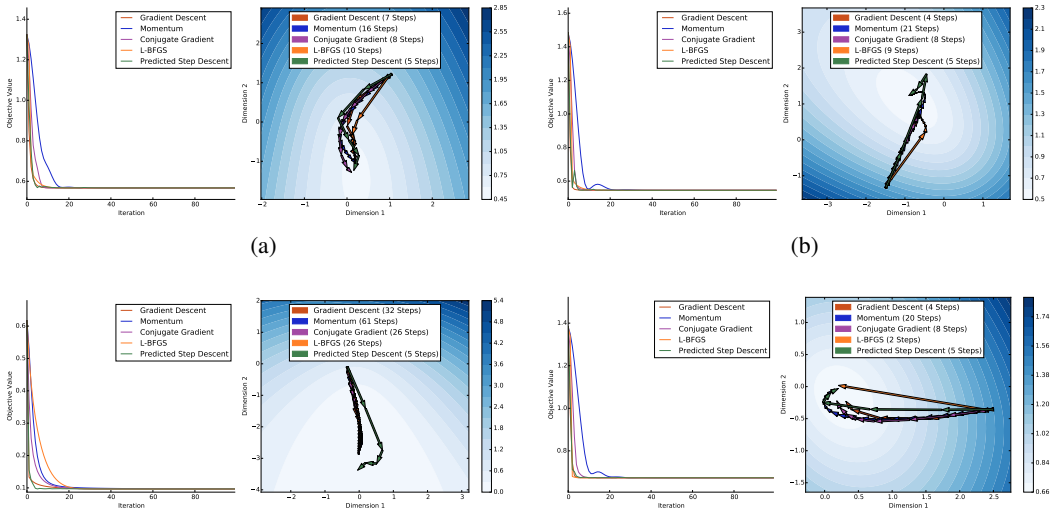


Figure 4: Objective values and trajectories produced by different algorithms on unseen random two-dimensional logistic regression problems. Each pair of plots corresponds to a different logistic regression problem. Objective values are shown on the vertical axis in the left plot and as contour levels in the right plot, where darker shading represents higher objective values. In the right plot, the axes represent the values of the iterates in each dimension and are of the same scale. Each arrow represents one iteration of an algorithm, whose tail and tip correspond to the preceding and subsequent iterates respectively. Best viewed in colour.

As shown, the learned algorithm exhibits some interesting behaviours. In Figure 4a, the learned algorithm does not take as large a step as L-BFGS initially, but takes larger steps than L-BFGS later on as it approaches the optimum. In other words, the learned algorithm appears to be not as greedy as L-BFGS. In Figures 4b and 4d, the learned algorithm initially overshoots, but appears to have learned how to recover while avoiding oscillations. In Figure 4c, the learned algorithm is able to make rapid progress despite vanishing gradients.

7 Conclusion

We presented a method for learning a better optimization algorithm. We formulated this as a reinforcement learning problem, in which any particular optimization algorithm can be represented as a policy. Learning an optimization algorithm then reduces to find the optimal policy. We used guided policy search for this purpose and trained optimizers for different classes of convex and non-convex objective functions. We demonstrated that the learned optimizer converges faster and/or reaches better optima than hand-engineered optimizers. We hope optimizers learned using the proposed approach can be used to solve various common classes of optimization problems more quickly and help accelerate the pace of research in science and engineering.

Acknowledgements

This work was supported by ONR MURI N00014-14-1-0671. Ke Li thanks the Natural Sciences and Engineering Research Council of Canada (NSERC) for fellowship support. The authors also thank Chelsea Finn for code and Pieter Abbeel, Sandy Huang and Zoe McCarthy for feedback. This research used the Savio computational cluster resource provided by the Berkeley Research Computing program at the University of California, Berkeley (supported by the UC Berkeley Chancellor, Vice Chancellor for Research, and Chief Information Officer).

REFERENCES

Yaser S Abu-Mostafa. A method for learning from hints. In *Advances in Neural Information Processing Systems*, pp. 73–80, 1993.

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

Jonathan Baxter, Rich Caruana, Tom Mitchell, Lorien Y Pratt, Daniel L Silver, and Sebastian Thrun. NIPS 1995 workshop on learning to learn: Knowledge consolidation and transfer in inductive systems. `https://web.archive.org/web/20000618135816/http://www.cs.cmu.edu/afs/cs.cmu.edu/user/caruana/pub/transfer.html`, 1995. Accessed: 2015-12-05.

Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000.

James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research*, 13(1):281–305, 2012.

James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in Neural Information Processing Systems*, pp. 2546–2554, 2011.

Matthieu Bray, Esther Koller-meier, Pascal Müller, Luc Van Gool, and Nicol N Schraudolph. 3d hand tracking by rapid stochastic gradient descent using a skinning model. In *1st European Conference on Visual Media Production (CVMP)*. Citeseer, 2004.

Pavel Brazdil, Christophe Giraud Carrier, Carlos Soares, and Ricardo Vilalta. *Metalearning: applications to data mining*. Springer Science & Business Media, 2008.

Pavel B Brazdil, Carlos Soares, and Joaquim Pinto Da Costa. Ranking learning algorithms: Using ibl and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003.

Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms*, pp. 183–187, 1985.

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

Justin Domke. Generic methods for optimization-based modeling. In *AISTATS*, volume 22, pp. 318–326, 2012.

Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Initializing bayesian hyperparameter optimization via meta-learning. In *AAAI*, pp. 1128–1135, 2015.

Chelsea Finn, Xin Yu Tan, Yan Duan, Trevor Darrell, Sergey Levine, and Pieter Abbeel. Learning visual feature spaces for robotic manipulation with deep spatial autoencoders. *arXiv preprint arXiv:1509.06113*, 2015.

Jie Fu, Zichuan Lin, Miao Liu, Nicholas Leonard, Jiashi Feng, and Tat-Seng Chua. Deep q-networks for accelerating the training of deep neural networks. *arXiv preprint arXiv:1606.01467*, 2016.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Karol Gregor and Yann LeCun. Learning fast approximations of sparse coding. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 399–406, 2010.

Weiqiao Han, Sergey Levine, and Pieter Abbeel. Learning compound multi-step controllers under unknown dynamics. In *International Conference on Intelligent Robots and Systems*, 2015.

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

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pp. 87–94. Springer, 2001.

Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pp. 507–523. Springer, 2011.

Sergey Levine and Pieter Abbeel. Learning neural network policies with guided policy search under unknown dynamics. In *Advances in Neural Information Processing Systems*, pp. 1071–1079, 2014.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *arXiv preprint arXiv:1504.00702*, 2015a.

Sergey Levine, Nolan Wagener, and Pieter Abbeel. Learning contact-rich manipulation skills with guided policy search. *arXiv preprint arXiv:1501.05611*, 2015b.

Percy Liang, Michael I Jordan, and Dan Klein. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 639–646, 2010.

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

Jonas Mockus, Vytautas Tiesis, and Antanas Zilinskas. The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2(117-129):2, 1978.

Paul L Ruvolo, Ian Fasel, and Javier R Movellan. Optimization on a budget: A reinforcement learning approach. In *Advances in Neural Information Processing Systems*, pp. 1385–1392, 2009.

Jürgen Schmidhuber. Optimal ordered problem solver. *Machine Learning*, 54(3):211–254, 2004.

Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pp. 2951–2959, 2012.

Pablo Sprechmann, Roee Litman, Tal Ben Yakar, Alexander M Bronstein, and Guillermo Sapiro. Supervised sparse analysis and synthesis operators. In *Advances in Neural Information Processing Systems*, pp. 908–916, 2013.

Kevin Swersky, Jasper Snoek, and Ryan P Adams. Multi-task bayesian optimization. In *Advances in neural information processing systems*, pp. 2004–2012, 2013.

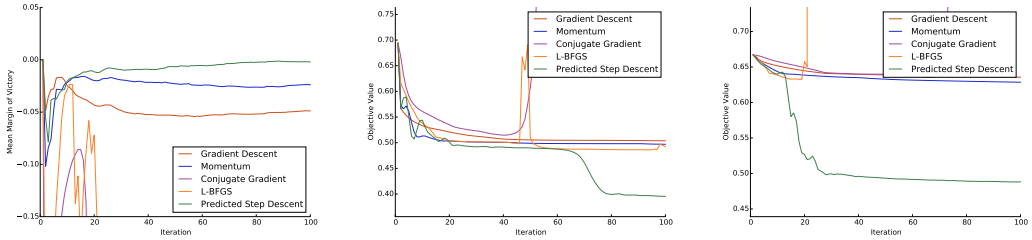
Sebastian Thrun and Lorien Pratt. *Learning to learn*. Springer Science & Business Media, 2012.

Ricardo Vilalta and Youssef Drissi. A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

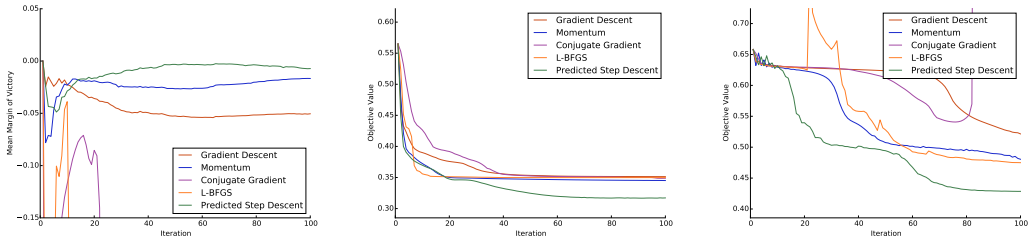
8 APPENDIX

8.1 TRANSFER TO OBJECTIVE FUNCTIONS DRAWN FROM DIFFERENT DISTRIBUTIONS

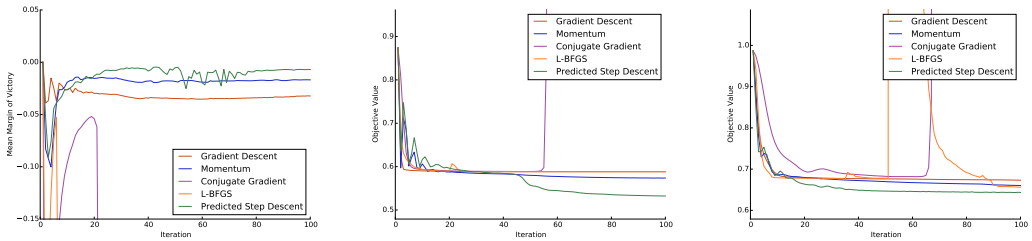
We evaluate the learned neural net optimizer, which is trained on neural net classification problems on data drawn from mixtures of four random Gaussians, on neural net classification problems on data drawn from mixtures of different numbers of random Gaussians. As the number of mixture components increases, the data distributions used at test time become more dissimilar from those seen during training. As shown in Figures 5, the learned optimizer seems to be fairly robust and performs reasonably well despite deviations of data distributions from those used for training.



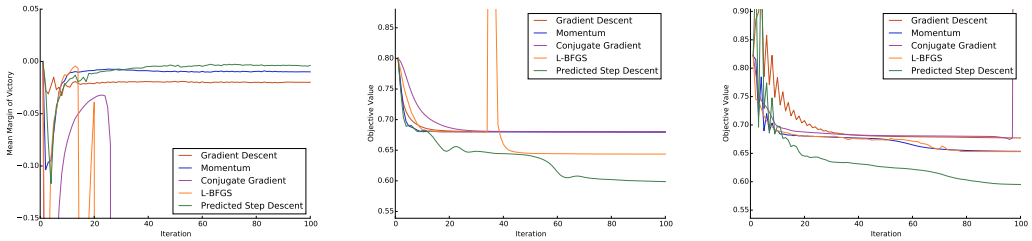
(a) Mixture of 5 Gaussians



(b) Mixture of 6 Gaussians



(c) Mixture of 8 Gaussians



(d) Mixture of 12 Gaussians

Figure 5: Performance of the learned neural net optimizer on classification problems on data drawn from mixtures of different numbers (5, 6, 8 and 12) of random Gaussians. The left column shows the mean margin of victory of each algorithm, and the middle and right columns show objective values achieved by each algorithm on two neural net classification problems from the test set. Best viewed in colour.