Rover Descent: Learning to optimize by learning to navigate on prototypical loss surfaces

Louis Faury $^{1,2,*}$ and Flavian Vasile$^1$

$^1$Criteo Research, Paris
$^2$Ecole Polytechnique Federale de Lausanne, Switzerland

* l.faury@criteo.com

Learning to optimize - the idea that we can learn from data algorithms that optimize a numerical criterion - has recently been at the heart of a growing number of research efforts. One of the most challenging issues within this approach is to learn a policy that is able to optimize over classes of functions that are fairly different from the ones that it was trained on. We propose a novel way of framing learning to optimize as a problem of learning a good navigation policy on a partially observable loss surface. To this end, we develop Rover Descent, a solution that allows us to learn a fairly broad optimization policy from training on a small set of prototypical two-dimensional surfaces that encompasses the classically hard cases such as valleys, plateaus, cliffs and saddles and by using strictly zero-order information. We show that, without having access to gradient or curvature information, we achieve state-of-the-art convergence speed on optimization problems not presented at training time such as the Rosenbrock function and other hard cases in two dimensions. We extend our framework to optimize over high dimensional landscapes, while still handling only two-dimensional local landscape information and show good preliminary results.

1 Introduction

Finding the minimizer $\theta^*$ of a function $f$ over some domain $\Omega$ is a recurrent problem in a large variety of engineering and scientific tasks. Instances of this problem appear in machine learning, optimal control, inventory management, portfolio optimization, and many other applications. This great diversity of problems has led over the years to the development of a large body of optimization algorithms, from very general to problem-specific ones.

Recently, the advent of deep learning led to the creation of several methods targeting high-dimensional, non-convex problems (the most famous ones being momentum [Nesterov, 1983], Adadelta [Zeiler, 2012] and Adam [Kingma and Ba, 2014]), now used as black-box algorithms by a majority of practitioners. Other attempts in this field use some additional problem-specific structure, like the work by [Martens, 2010] that leverages fast multiplication by the Hessian to yield better performing, though computationally demanding, optimization policies. A common point to all these algorithms is that they leverage human-based understanding of loss surfaces, and usually require tuning of hyper-parameters to achieve state-of-the-art performance. This tuning process can sometimes reveal mysterious behavior of the handled optimizers, making it reserved to human experts or the subject of a long and tedious search. Also, the process results in a static optimizer which excels at the specific task, but is likely to perform poorly on others.

If the limitations of hand-designed algorithms come from poor human understanding of high-dimensional loss landscapes, it is natural to ask what machine learning can do for optimization algorithms. Recently, [Andrychowicz et al., 2016] and [Li and Malik, 2016] both introduced two frameworks for learning optimization algorithms. While the former proposes to learn task-specific optimizers, the latter aims to produce for task-independent optimization policies. While in the most general case this is bound to fail - as suggested by the "No Free Lunch" theorem for combinatorial optimization by [Wolpert and Macready, 1997] - we also believe that robust techniques can be learnt from data to be applied on a great variety of natural problems.

Most optimization algorithms can be framed, for a given objective function $f$ and a current iterate $\theta_i$, as finding an appropriate update $\Delta \theta_i$. This update can for instance depend on past gradient information,
rescaled gradient using curvature information or many other features. In a general manner, we can write \( \Delta \theta_i = \phi(\theta_i, h(f, \theta_{i-1}, \ldots, \theta_0), \xi) \) where \( h(\cdot) \) denotes the set of features accumulated during the optimization procedure, and \( \xi \) denotes the optimization hyper-parameters.

In our approach, we aim to bypass computing gradient and curvature information and learn the optimization features directly from data with the goal of obtaining local state descriptors that could outperform classical features in terms of generalization on unseen loss functions and input data distributions. In this vein, we draw an analogy between learning an optimization algorithm and learning a navigation policy while having access to raw local observations of the landscape, which is also the inspiration for the name of our method, Rover Descent. Our algorithm contains three chained predictors that compute the angle of the move, the magnitude of the move (e.g. learning rate) and the resolution of the grid of the zero-order samples at the landing point. We train our navigation agent on hard "prototypical 2D surfaces" in order to make sure we develop feature detectors and subsequent policies that will be able to lead to good decisions in difficult areas of the loss function. We pose both the learning rate and resolution predictor as Reinforcement Learning problems and introduce a reward shaping formula that allows us to learn from functions with different magnitude and from multiple proto-families, a factor that proved to be very important in being able to generalize on many different types of evaluation functions.

We show that this setup leads to very good convergence speeds both in two and higher dimensions, on evaluation functions that are not presented at training time. For a zero-order optimization algorithm, the convergence performance is surprisingly good, leading to results comparative or better than the best optimizer for the task (either first or second-order based).

In conclusion, we believe that our main contributions are the following: framing the problem learning to optimize as a navigation task, proposing a zero-order information-based learning architecture, coupled with a proper training procedure on prototypical 2D surfaces and a reward shaping formula and showing experimentally that it can match/outperform first and second order techniques on meta-generalization tasks.

The rest of the paper is organized as follows. We first give a brief summary of past and recent related work in the field of learning to learn and learning to optimize, and position our approach with respect to other existing work in Section 2. We then develop in 3.2 our approach in the two-dimensional case, before extending it to a higher dimensional setting in 3.3. We show experimental results in Section 4 justifying our approach in a variety of setups. We finally develop potential ideas for future work in Section 5.

2 Related work

Learning to learn (or meta-learning) is not a recent idea. [Schmidhuber, 1987] thought of a Recurrent Neural Network (RNN) able to modify its own weights, building a fully differentiable system allowing the training to be learned by gradient descent. [Hochreiter et al., 2001] proposed to discover optimizers by gradient descent, optimizing RNNs modeling the optimization sequence with a learning signal emerging from backpropagation on a first network.

Recently, some meta-learning tentatives have shown great progress in different optimization fields. Using gradient statistics as an input for a recurrent neural net, [Li and Malik, 2017] were able to reinforce learn a policy effective for training deep neural networks. In [Andrychowicz et al., 2016], the authors show that when leveraging first-order information one could learn by gradient-descent optimizers that outperforms current state of the art of existing problems - however only when the meta-train dataset is made of the same class of problem. When confronted to a different class of functions, the meta-learner is unable to infer efficient optimization moves or completely fails. With the same idea of using gradient-descent for training the optimizer, [Chen et al., 2017] use zeroth order information in order to learn an optimizer for the Bayesian optimization setting - using only noisy zeroth order oracles.

However, one could think that by showing enough examples to a meta-learner (namely made up of instances where traditional optimizers reach their limits), and adapting its structure to cover larger fields of function class, it could adapt to unknown loss landscapes. This idea was exploited by [Wichrowska et al., 2017], who manage to learn optimizers that generalizes to completely unseen data, while still being able to scale up to high-dimensional problems. Their process namely involves training by gradient descent hierarchical RNNs and showing it a great variety of examples. However, their optimizer’s structure remains quite complicated, and doesn’t provide human-level understanding of the features leveraged by the meta-learner. We believe that a more down-to-earth approach could enable us to understand what the network is learning, while still being effective on a large class of functions, when trained on a selected number of meta-examples.
3 Our approach

3.1 Intuition

Hand-designed optimization algorithms, once well tuned for a given problem, can already provide good baselines for providing optimization updates. However, they are susceptible to some degeneracies in the loss surface they are exploring (valley for gradient descent and many of its variants, saddle-point for curvature-based optimizer). Many efforts were recently put in overcoming these difficulties, or in trying to revise some older ideas in the optimization community to achieve efficient minimization of large-scale non-convex functions (a good example of this can be found in [Hazan et al., 2016]). Deep-learning provides a good benchmarking environment for such problems and led to many innovations. We want to consider such strategies as teachers for our learning algorithm. As we noted before, it often happens that these common optimizers each have a nemesis landscape on which their improvement can be very slow or even inexistent.

In our paper, we propose framing the problem of Learning to Optimize as a problem of navigation on the partially observable error surface. The error surface is defined by the values of the loss function taken over the range of its inputs. In this framework, the optimiser is an agent that starting from the initial point, attempts to reach the lowest point on the surface with the smallest number of actions (where an action is a move to an arbitrary point on the landscape), while observing only a small grid of points sampled from the loss surface, centered at its current location. Our goal is to learn the navigation policy that maps the current state of the agent to a move on the surface.

To this end, we decided to divide the architecture of our agent in three sequential modules: the normalized update direction predictor $\Delta$, that takes the grid of observed loss function values and predicts the angle of the update, the learning rate predictor that predicts the magnitude of the update $\alpha$, and the resolution predictor, that predicts the scale $\delta$ of the grid of observations around the landing point. This choice is motivated by our intuition that these steps can be approached in a hierarchical way (first choose a direction, then a step size accordingly for instance) and therefore might involve different training methods and procedures. Furthermore, each of the modules can act as a correction factor on the other two modules. For example, if the update angle is not correct, the learning rate module can compensate by making the move very small and the resolution module can zoom out/in to make the next angle prediction task easier.

3.2 Architecture: The two-dimensional case

In the following subsection, we consider the simple case $d = 2$ to develop our experimental set-up. A generalization for higher dimensions can be found in the next subsection. Figure 1 presents the architecture of this decomposition.

3.2.1 Choosing the Proto Landscapes and the input representation

Proto Landscapes Because our end goal is to be able to optimize high-dimensional loss landscapes, our idea was to select a small but sufficient number of unitary landscapes that are frequent in this setting. More precisely, we decided to target surface degeneracies that are common when learning the weights of
deep neural networks. These namely include narrow valleys, plateaus, but also cliffs ([Bengio et al., 1994]) and saddles ([Dauphin et al., 2014]). We also considered it useful to add quadratic bowls to that list, to provide simpler and saner landscapes. Figure 3 provides a visualisation of each of these landscapes. Interestingly enough, all of these landscapes were listed in [Schaul et al., 2013], who are providing a collection of unit tests for optimization. In the line of this work, we estimate that learning a optimizer over such landscapes could result in a robust algorithm. Also, because it is frequent in real world applications to only have access to noisy samples of the function we wish to optimize, our framework should therefore provide noisy version of the landscape described hereinbefore.

**Input design** We decided to retain two different teachers for our experiment: gradient-descent and Newton descent (see [Nocedal and Wright, 2006] for complete details on these optimizers). While gradient descend can escape non-strict saddle points but shows shattering behavior inside valleys (see Figure 2), second-order methods can leverage curvature to make quick progress inside valleys. However, saddles are attraction points for such methods. Finite difference provides an easy way to approximately reconstruct both first and second order informations from zeroth order sample of a function. However, the precision of finite difference can be severely impacted by noisy oracles, although this can be alleviated by a pre-filtering of the function samples (like low-pass filtering for removing white noise).

![Figure 2: Left: gradient descent has a shattering behavior in narrow valleys. Right: saddle points are attractors for Newton descent.](image)

Let \( f \) the loss landscape we are optimizing, mapping \( \Omega_f \subset \mathbb{R}^d \) into \( \mathbb{R} \), and \( \theta \in \Omega_f \). A natural way to describe the surroundings of \( \theta \) is to sample a grid centered on \( \theta \). Given a budget of \( n^2 \) samples and a resolution \( \delta \), we note \( s_n^\delta(f,\theta) \) the resulting two-dimensional grid.

\[
s_n^\delta(f,\theta) \triangleq \left( f\left( \theta + \delta \left( \frac{i-n/2}{n}, \frac{j-n/2}{n} \right) \right) \right)_{i,j \in \{1,...,n\}^2}
\]  

This state representation allows to have a human understandable input to our model, represent the surroundings of the current iterate and approximate the inputs taken by gradient descent and Newton descend via finite difference. Also, pre-filtering can be efficiently applied by convolutions. For this state representation to represent compactly various functions, independent of their magnitude, we linearly rescale it to take its values in \([0,1]\].

It is important to note that such an input becomes extremely expensive to compute as the dimensionality of the problem grows, as the size of \( s_n^\delta(\cdot) \) grows exponentially with \( d \). Therefore, we will use this solution for \( d = 2 \), and discuss different ways of scaling to higher dimension in 3.3.

One could argue that using noisy zeroth order oracle for optimization is uncompetitive compared to higher order methods. Indeed, the study of convergence rates and lower bounds for convex optimization problem show the superiority of first-order oracles over single zeroth-order function evaluation ([Nemirovskii et al., 1983]). However, it was proven in [Duchi et al., 2015] that by using only two function evaluation, the sample complexity of the latter type of algorithms could compete with the former, up to a low-order polynomial of the dimension factor (in the convex case). Because we use many of such samples, we are confident that we will be competitive against higher order oracles.

**Landscapes generation** We generate random instances of five landscapes: saddles, valleys, plateaus, cliffs and quadratic bowls. The quadratic bowls are generated by sampling random matrices \( A \) and vectors \( b \) and aggregating them in a quadratic loss \( \|Ax - b\|^2_2 \). The other four landscapes are generated by creating random gaussian fields with Mahalanobis-norm covariance functions.
More precisely, we carefully sample a given number of points $x$ which are attributed a random value $f(x)$. We then sample a covariance function $k(x, x') = \frac{1}{2}x^T S^{-1} x$ with $S$ being a randomly generated definite positive matrix, carefully set to generate the targeted landscape. For the sake of completeness, we detail the generation of meta-train landscape in Appendix 6.1.

We therefore create a class of function $F_{\text{train}}$ from which we can sample random instances of the different targeted landscapes. We also add randomness inside each of these instances by not always taking the mean of the generated random fields, but by sampling inside the resulting distribution over space. Figure 3 displays 3-dimensional views of sample of each of these instances (the functions generated here are obtained by taking the mean of the random fields for visualisation purposes).

![Figure 3: Instances of $F_{\text{train}}$. In order: quadratic bowl, valley, (plateau+cliff), saddle. Best viewed in color.](image)

### 3.2.2 Learning the update direction/angle

The first step of our three-step optimizer is to determine a good direction of update, given a grid of samples $s^\theta_n(f, \theta)$. In light of the previous discussion, we decided to learn this direction by imitation learning, provided the two teachers we selected earlier: gradient descent ($\Delta \theta \propto -\nabla_\theta f(\theta)$) and Newton descent ($\Delta \theta \propto -\nabla_\theta^2 f(\theta)^{-1} \nabla_\theta f(\theta)$). The field of imitation learning is fairly large, though dominated by two antagonist approaches: behavioral cloning and inverse reinforcement learning. The latter involves recovering the cost function a teacher or expert is trying to minimize, while the former involves training a complex model (usually a deep neural network) in a supervised fashion so that it mimics a teacher. Thorough details on both these methods, as well as a complete survey of the field of imitation learning can be found in [Billard et al., 2016].

Behavioral cloning, while being straight forward and simple to implement, is known to require a large amount of data and to be prone to compounding errors, leading to divergence between the teacher’s and the imitation followed paths.

On the other side, inverse reinforcement learning allows the imitator to interact with the environment, and fit its behavior over whole trajectories (therefore is not affected by the compounding error issue). However, it often implies using reinforcement learning in a inner loop, making this technique rather costly to use. In our set-up, we decided to use a behavioral cloning approach. We can indeed easily generate large amounts of training data, and are not trying to fit the entire teacher behavior but only a subpart - the direction, not the step-size.

We follow a simple procedure to compute our meta-training dataset detailed in Figure 4.

At each step of the optimization procedure (where we follow the best out of the teachers), we record the state-representation alongside two opposite directions: the optimal one $d_\ast$ and a set of opposite randomly generated ones $\tilde{d}_\ast$ (sampled to lie in the half-space defined by $\{d | d^T d_\ast < 0\}$). The expert move $d_\ast$ and its negative counterparts $\tilde{d}_\ast$ are normalized to create two actions $a_\ast = d_\ast/\|d_\ast\|_2$ and $\tilde{a}_\ast$ similarly. We create two state-action pairs with respective label $t = 1$ and $t = 0$, corresponding to a positive and a negative sample. In practice, we sampled $5 \cdot 10^4$ functions from $F_{\text{train}}$ and let the optimization procedure run for 10 steps on each functions, creating $10^6$ (state,action,label) tuples to train on, stored in $D_{\text{train}}$.

To fit the resulting (state-action) pairs, we designed a simple neural network made of two convolutional layers followed by two fully connected layers. The output layer maps the last hidden layers in $\mathbb{R}^2$. For a given grid of sample $s$, we denote $y(\omega, s)$ the output of this model, parametrized by the weights $\omega$ of the network. We use batch-normalization layers after the convolutional layers. No drop-out or pooling was
used. The model was trained to minimize the cross-entropy loss:

$$J(D_{\text{train}}, \omega) = \sum_{(s,a) \in D_{\text{train}}} \left[ t \log \{ \sigma(y(s, a)) \} + (1 - t) \log \{ 1 - \sigma(y(s, a)) \} \right]$$

(2)

with $\sigma(y(s), a) = \frac{1}{1 + e^{-y(s, a)}}$. The objective is to learn to correlate the output of the model when the action $a$ has a positive label (it was sampled from the best teacher). The idea of storing negative versions of that optimal action can be understood as negative sampling. We found that this approach, over the other ones we tried, lead to better performances while greatly reducing overfitting. Because we only want to use this model as an angle predictor, we will use a normalized version of the output: $\Delta(s) = y(s)/\|y(s)\|_2$.

### 3.2.3 Learning the step-size and the resolution

At this point, we have learnt a good angle predictor. We now want to learn two new behaviors: the step-size to apply to the update, as well as the next resolution of the sample grid. Learning the step-size is obviously crucial for the optimization step. Learning the resolution is also extremely important: far from an optimum, we’d like to zoom-out to get a better understanding of the landscape. Close to an optimum, we expect an efficient system to zoom in to refine its estimation of the localization of the optimal point. Those two behaviors can’t be learnt efficiently from a teacher (line-search is an unfairly good teacher for the step-size, and we simply don’t have available a hand-designed teacher for the resolution).

Reinforcement learning is a framework in which an agent learns its actions from interaction with its environment. The environment generates scalar values called rewards, that the agent is seeking to maximize over time. The environment is modeled as a partially observable Markov decision process (POMDP), defined to be the tuple \((O, S, A, p_0, p, q, r)\) where \(O\) is the set of observations, \(S\) the set of states and \(A\) the set of actions. \(p_0(s)\) is the initial probability distribution over the states, \(p(s'|s, a)\) the transition model, \(p(a|s)\) the distribution of an observation conditionally to a state and \(r : S \rightarrow \mathbb{R}\) a function that assigns a reward to each state. The objective is to learn a policy \(\pi(a|s) : S \rightarrow A\) providing the probability of choosing action \(a\) in state \(s\). This policy should maximize the discounted expected return \(\bar{R}\):

$$\bar{R} = \mathbb{E}_\rho \left[ \sum_{t=0}^T \gamma^t r(s_t) \right]$$

(3)

where \(\gamma \in (0, 1)\) is a discount factor allowing the agent to be more sensitive to rewards it will get in a close future, and the expectation being taken with respect to the state-action distribution \(\rho\). A complete introduction to the reinforcement learning framework can be found in [Sutton and Barto, 1998].

Policy search is a family of algorithm that directly search in the policy space for \(\pi^* = \arg\max_{\pi} \{ R \}\). To make this search tractable, \(\pi\) is usually tied to some parametrized family. A popular algorithm to perform that search is the Deterministic Policy Gradient [Silver et al., 2014] where we learn a deterministic parametrized policy \(\pi_\eta(a|s) = \mu(\eta, s)\) in a fully observable Markov decision process \((O = S)\). The system is composed of two entities, an actor and a critic. The critic, parametrized by \(\omega\), has the role to evaluate the Q-values (expected return when taking an action \(a\) at a state \(s\)) of the current policy induced by the actor (parametrized by \(\eta\)). As it is common in actor-critic approaches, the critic is updated by batch of logged experience to minimize the squared temporal difference (TD) error \((r_t + \gamma Q_\omega(s_{t+1}, a_{t+1}) - Q_\omega(s_t, a_t))^2\). The actor’s parameters are updated in the direction that maximizes the Q-values for a batch of logged states: \(\Delta \eta \propto \nabla_\eta Q_\omega(s, a)^T \nabla_\omega \mu(\eta, s)\). [Lillicrap et al., 2015] applied this algorithm to deep neural networks as function approximators, using techniques that were proven successful in deep Q-learning, like target networks and experience replay. [Heess et al., 2015] also extended this approach for POMDP, where it is useful to use Recurrent Neural Networks as models for the policy.

We consider the following environment for our problem. We let, for a given loss function \(f\), the full state space \(S_f = \{\theta, \alpha, \delta\}\) and the observations \(O_f = (s^o_\theta(f, x))\). The agent hence only has access to the current grid of samples around \(\theta\) with resolution \(\delta\), but not to the current iterate position \(\theta\), the current-step size \(\alpha\) or the current resolution \(\delta\). The idea behind this is to be able to generalize to unseen landscapes, and be robust to transformations such as rescaling or translations. The only events that should impact the agent’s behavior is a sharp change in the neighboring landscape around the current iterate. The action space is set to be \(A = \{\Delta \alpha, \Delta \delta\} \subseteq [-0.5, 1]^2\) which constitutes the update rate of
the step-size and the resolution. We consider deterministic transitions:

\[
\begin{align*}
\theta_{t+1} &= \theta_{t} + \alpha_t \Delta(s^n_t(f, \theta_t)) \\
\alpha_{t+1} &= \alpha_t (1 + \Delta \alpha_t) \\
\delta_{t+1} &= \delta_t (1 + \Delta \delta_t)
\end{align*}
\]

where the current iterate \(\theta_t\) is updated along the direction \(\Delta(s^n_t(f, \theta_t))\) with step-size \(\alpha_t\). We have several options for the reward function. One possibility would be to consider a budgeted optimization scheme, with reward \(r(s) = -f(\theta_t) 1_{t=T}\) (the reward is only given by the final value of the function at the last step). In this case, the reward is rather sparse, and leads the trajectory search in ambiguous ways. We preferred another solution, where the whole trajectory of the agent over the landscapes is evaluated:

\(r_f(s) = -f(\theta_t)\). This leads the policy search to optimize for following return:

\[
R_f = -\sum_{t=0}^{T} \gamma^t f(\theta_t)
\]

(5)

Note that for \(\gamma = 1\) this leads us to optimize over the same criterion that [Li and Malik, 2017] and [Andrychowicz et al., 2016] (in the former, the author call this the meta-loss).

It is important to note that the previously described POMDP - that we will denote as \(\mathcal{M}_f\) - is parametrized by a function \(f\) sampled inside \(\mathcal{F}_{\text{train}}\). This induces a distribution \(\mathcal{M}_{\text{train}}\) over POMDPs. In the following experiments, we won’t make that distinction and train a single parametrized policy on the resulting POMDP distribution - that implies that every new episode is generated with \(\mathcal{M}_f \sim \mathcal{M}_{\text{train}}\). This induces a difficulty over the learning task: both the transitions and the reward defined in (4) and (5) change between every episode. To help the agent figure out optimal moves, we change the reward function so it becomes insensitive to the magnitude of the sampled function \(f\) and the position of the initial iterate \(\theta_0\):

\[
r_f(s_t) = f(\theta_t) - f(\theta^*_f)\]

(6)

with \(\theta^*_f = \text{argmin}\{f(\theta)\}\). To also help the agent optimize over long trajectories where the magnitude of \(f(\theta_0)\) largely surpasses \(f(\theta^*_f)\), we finally picked the following reward function:

\[
r_f(s_t) = \frac{f(\theta_t) - f(\theta^*_f)}{\bar{f}_k - f(\theta^*_f)}
\]

(7)

with \(\bar{f}_k\) being the mean value of the objective function over the last \(k\) iterates (we found that in our set-up, \(k = 5\) provided good results).

We model the agent policy by a recurrent neural network, made up of 2 convolutional layers, followed by a Long-Short Term-Memory cell (LSTM, introduced by [Hochreiter and Schmidhuber, 1997]), followed itself by two hidden layers. The critic is also model by a similar network, and both were trained using the DPG algorithm. During training, we sample \(f \sim \mathcal{F}_{\text{train}}\) at the beginning of each episode. The initial iterate is randomly sampled in the landscape so that it is far away enough from the optimum of the loss function. The episode is run for a fixed horizon \(T = 30\) and we fix the discount factor \(\gamma\) to 1.

### 3.3 Architecture for \(d > 2\)

The idea of using grid samples \(s^n_i(f, \theta)\) can’t be exploited in high-dimensional problems as its size grows exponentially with \(d\). To extend our framework for \(d > 2\), we consider the following set-up: let \(f : \mathbb{R}^d \rightarrow \mathbb{R}\) and \(\theta\) the initial iterate. We note \(s(f, \theta, i, j)\) the vector that contains the two-dimensional grid sampled at \(\theta\) along the dimensions \(i\) and \(j\). In other words, with \(\delta_d\), the \(d\)-dimensional vector whose entries are all 0 but the \(i\)th one that is set to 1, and \(E_{i,j} = (\delta_i, \delta_j) \in \mathcal{M}_{d,2}[^d(\mathbb{R})]\), we note:

\[
s^n_i(f, \theta, i, j) = s^n_i(f, E_{i,j}^d, \theta).
\]

By considering all pairs of dimensions, we can compute \(d(d-1)/2\) of such grids, leading to the prediction of as many angles \(\Delta_{i,j}(f, \theta) = \Delta(s^n_{i,j}(f, \theta, i, j), \text{step-size updates } \Delta \alpha_{i,j}\) and resolution updates \(\Delta \delta_{i,j}\) - all predicted with the models trained in the two dimensional case. Therefore, if we keep record of all step-size and resolution for every pair of dimension \((i, j)\) we can compute \(d(d-1)/2\) updates \(\Delta \theta_{i,j} = \alpha_{i,j} \Delta_{i,j}(f, \theta)\). We can consider each of those output like the \(d(d-1)/2\) two-dimensional projections of
the true $d$-dimensional update $\Delta \theta$ so that $\Delta \theta_{i,j} = E_{i,j}^T \Delta \theta$. We can therefore try to retrieve $\Delta \theta$ by a least-square approach, and find $\Delta \hat{\theta}$:

$$
\Delta \hat{\theta} \triangleq \arg\min_{\delta \theta} \sum_{1 \leq i < j \leq d} (E_{i,j}^T \delta \theta - \Delta \theta_{i,j})^2
$$

(8)

Solving this equation leads to the analytical expression:

$$
\Delta \hat{\theta} = \frac{1}{d-1} \sum_{1 \leq i < j \leq d} \alpha_{i,j} E_{i,j} \Delta_{i,j}(f, \theta)
$$

(9)

Each pair of dimension is therefore attached its step-size $\alpha_{i,j}$ and resolution $\delta_{i,j}$, that are updated by running the associated two-dimensional grid through the system described in 3.2.3. This computation requires maintaining $d(d-1)/2$ learning rates and resolutions, and computing as many grid samples. Because of this quadratical growth with the dimension, this can lead to clock-time issues for large values of $d$. A simple way round this problem is to uniformly sample $k < d$ pairs of dimensions, compute $\Delta \hat{\theta}$ based only on these $k$ pairs and update their corresponding learning rates and resolutions. If we note $\Theta_k$ the set of $k$ pairs we sampled:

$$
\Delta \hat{\theta}_k = \frac{1}{k-1} \sum_{(i,j) \in \Theta_k} \alpha_{i,j} E_{i,j} \Delta_{i,j}(f, \theta)
$$

(10)

Figure sums up the full optimization procedure for $d > 2$.

Input: Dimension $n$, budget $k$, function $f$, initial step-size $\alpha_0$, initial resolution $\delta_0$, initial iterate $\theta_0$.

Initialize: step-size and resolution arrays with respectively $\alpha_0$ and $\delta_0$.

Repeat:

(a) Sample $k$ pairs $(i,j)$ in $(1 \leq i < j \leq n)$ uniformly at random.

(b) Compute $\Delta \hat{\theta}_k$ using (10).

(c) For each pair in $(i,j) \in \Theta_k$, update the step-size $\alpha_{i,j}$ and resolution $\delta_{i,j}$.

Figure 4: Optimization procedure using the two-dimensional learnt optimizer for dimensions $d > 2$.

4 Results

4.1 Two-dimensional experiments

We first want to evaluate the optimizer resulting from the model we introduced on $F_{train}$ to evaluate its behavior on known landscapes. This already can be seen as some kind of meta-testing on some hold-out since we only sample in $F_{train}$, which contain an infinite number of functions. Therefore, we can assume that whenever we sample in $F_{train}$, we obtain a function the optimizer have not seen during training.

We follow a simple procedure: we sample $f \in F_{train}$, an initial point $\theta_0 \in \Omega_0$, and sample uniformly at random an initial step-size and an initial resolution inside the distribution used at meta-training time. We then add a small perturbation to the initial iterate and run an optimization run with a fixed horizon. This procedure was repeated many times to evaluate the global sensitivity of our algorithm to the position of the first iterate. To compare its performance with a broad variety of optimizers, we decided to evaluate with the same procedure a collection of optimization algorithm that include: gradient descent, Nesterov accelerated gradient descent, Newton Descent, Covariance Matrix Adaptation [Hansen, 2016] and the Nelder-Mead method. The results are regrouped in Figure 5. The lines represent the mean trajectory of each optimizer, while the shaded areas represent the envelope of all its trajectories (that were generated from noisy versions of the initial iterate). The results, shown here for a single function $f$ and iterate point $\theta_0$ are consistent in our experiments - that we have learnt to compete with a wide variety of hand-designed algorithms, and that our optimizer is fairly independent of the initial value set for $\alpha_0$ and $\delta_0$. 
The hyper-parameters of hand-designed are modified at each time to perform as well as possible on the whole modality of $F_{\text{train}}$, we are testing on. This means that our learnt optimizer sometimes compete with unfairly good hand-design algorithms (like Newton descent on a quadratic loss, that hits the optimum after just one iteration). The appearing lack of trajectory envelope that can sometime be seen is due to the fact that the perturbation on the initial point sometimes have to be reduced for visualisation purposes.

To evaluate the meta-generalization abilities of our learnt optimizer, we also evaluate it on a two-dimensional meta-testing dataset $F_{\text{test}}$. We selected various two-dimensional optimization problem known to be challenging for general optimization methods. The complete lists contains Rosenbrock, Ackley, Rastrigin, Maccornick, Beale and Styblinski’s function, which literal expressions and surface plots can be found in Appendix 6.2. It is important to note that none of these landscapes were seen by the optimizer during its training.

For each of those functions, we selected a starting point that constitute a challenge for all compared optimizers (also indicated on the surface plot in Appendix 6.2). We then followed the previously described procedure. The results are displayed in Figure 6, and remains consistent when changing the starting point for each of those meta-test function. Our learnt optimizer can generalize to new landscapes, even multimodal ones, and compete with a wide variety of optimizers. Note that except CMA-ES, no hand-designed optimizer in our collection was able to solve each one of the toy problems, while our learnt optimizer without tuned hyper-parameters managed to find each global minimum. This proves the robustness of our learnt optimizer with respect to the hyper-parameters that constitute the initial resolution and initial learning rate.

4.2 High-dimensional experiments

We now test the procedure described in Section 3.3 for problem of dimensions $d > 2$. To do so, we generate a random binary classification tasks in high dimension, according to the framework described
in [Guyon, 2003]. We want to optimize over the cross-entropy loss induced by this dataset. We therefore sample an initial iterate $\theta_0$, an initial learning rate and an initial resolution for our optimizer, and launch an optimization run. We tests against two fairly good optimizers for this tasks: tuned gradient descent and Newton descent. We consider using a fixed budget $k = 10$ of dimensions we can sample at each iteration. The results for three different randomly generated dataset of different dimensions are presented in Figure 7.

The results presented here for $d > 2$ are consistent in our experiment - that is that our procedure for $d > 2$ competes with tuned optimizers that use respectively first and second order information. However, one major downside of our optimizer is clock-time performances - one optimization in this simple set-up can take up to a minute for $d > 50$, against a few seconds for gradient descent on the machine used for our experiments. Also, its performance is impacted by the under sampling that happens when the budget $k$ is strictly less than the actual dimension $d$. Directions for future work using this approach would be to intelligently sample the pairs of dimensions we sample at each iterations, maybe through a learnt behavior.
5 Conclusion

We introduced a new framework in order to achieve meta-generalization when learning to optimize. By combining tools from imitation learning and reinforcement learning, and defining the meta-dataset as a small set of prototypical functions that frequently appear in optimization problems, we were able to learn an optimization algorithm that generalizes well to unseen loss landscapes. Though this learnt optimizer is dependent on the setting of some hyper-parameters, we showed that their tuning doesn’t have a meaningful impact on performance, as the learnt optimizer is quite robust and can recover from bad initializations.

In future work, we plan on running our framework on more complex, high-dimensional non-convex problems - such as the training of deep neural networks. Also, we can imagine controlling even more the meta-train dataset in order to scan for such complicated landscapes. For instance, following a leave-one-out procedure, we could try to estimate the prototypical landscapes that are needed by a learnt optimizer to train a deep neural network, and therefore make further assumptions over the composition of the related loss landscapes. Furthermore, we wish to develop adaptive methods of dimension sampling in order to make our optimizer more efficient in very high dimensional problems. A more thorough evaluation of its performance will then involve testing it against other baselines that were recently develop in the learning to learn community to optimize deep neural networks.
References

[Andrychowicz et al., 2016] Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In Advances in Neural Information Processing Systems, pages 3981–3989.

[Bengio et al., 1994] Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE transactions on neural networks.

[Billard et al., 2016] Billard, A. G., Calinon, S., and Dillmann, R. (2016). Learning from Humans, pages 1995–2014. Springer International Publishing, Cham.

[Chen et al., 2017] Chen, Y., Hoffman, M. W., Colmenarejo, S. G., Denil, M., Lillicrap, T. P., Botvinick, M., and Freitas, N. (2017). Learning to learn without gradient descent by gradient descent. In International Conference on Machine Learning.

[Dauphin et al., 2014] Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In Advances in neural information processing systems.

[Duchi et al., 2015] Duchi, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., and Bengio, Y. (2015). On graduated optimization for stochastic non-convex problems. In International Conference on Machine Learning.

[Heess et al., 2015] Heess, N., Hunt, J. J., Lillicrap, T. P., and Silver, D. (2015). Memory-based control with recurrent neural networks. arXiv preprint arXiv:1512.04455.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S., and Schmidhuber, J. (1997). Long short-term memory. Neural computation.

[Hochreiter et al., 2001] Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to learn using gradient descent. In International Conference on Artificial Neural Networks.

[Kingma and Ba, 2014] Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980.

[Li and Malik, 2016] Li, K. and Malik, J. (2016). Learning to optimize. arXiv preprint arXiv:1606.01885.

[Li and Malik, 2017] Li, K. and Malik, J. (2017). Learning to optimize neural nets. arXiv preprint arXiv:1703.00441.

[Lillicrap et al., 2015] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.

[Martens, 2010] Martens, J. (2010). Deep learning via Hessian-free optimization. In ICML.

[Nemirovskii et al., 1983] Nemirovskii, A., Yudin, D. B., and Dawson, E. R. (1983). Problem complexity and method efficiency in optimization.

[Nesterov, 1983] Nesterov, Y. (1983). A method of solving a convex programming problem with convergence rate o (1/k2). In Soviet Mathematics Doklady, volume 27, pages 372–376.

[Nocedal and Wright, 2006] Nocedal, J. and Wright, S. (2006). Numerical optimization. Springer Science & Business Media.

[Rasmussen and Williams, 2006] Rasmussen, C. E. and Williams, C. K. (2006). Gaussian processes for machine learning, volume 1. MIT press Cambridge.
[Schaul et al., 2013] Schaul, T., Antonoglou, I., and Silver, D. (2013). Unit tests for stochastic optimization. arXiv preprint arXiv:1312.6055.

[Schmidhuber, 1987] Schmidhuber, J. (1987). Evolutionary principles in self-referential learning. on learning now to learn: The meta-meta-meta...-hook. Master’s thesis.

[Silver et al., 2014] Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. In Proceedings of the 31st International Conference on Machine Learning (ICML-14).

[Sutton and Barto, 1998] Sutton, R. S. and Barto, A. G. (1998). Reinforcement learning: An introduction. MIT press Cambridge.

[Wichrowska et al., 2017] Wichrowska, O., Maheswaranathan, N., Hoffman, M. W., Colmenarejo, S. G., Denil, M., de Freitas, N., and Sohl-Dickstein, J. (2017). Learned optimizers that scale and generalize. CoRR.

[Wolpert and Macready, 1997] Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. IEEE transactions on evolutionary computation.

[Zeiler, 2012] Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. arXiv preprint arXiv:1212.5701.
We then sample a positive definite scaling matrix $X$. It starts by carefully sampling a collections of points $\mathcal{X} = \{x_1, \ldots, x_k\}$ and their associated values $\mathcal{V} = \{v_1, \ldots, v_k\}$ to create a specifically targeted landscape. We then sample a positive definite scaling matrix $S$ for the normalized Gaussian covariance function:

$$k(x, x') = \frac{1}{|2\pi S|^{1/2}} \exp \left( -\frac{1}{2} x^T S^{-1} x' \right)$$  \hspace{1cm} (11)

Once these steps are completed, and for $F(x) \triangleq (f(x), v_1, \ldots, v_k)$, we make a Gaussian hypothesis over the joint distribution:

$$p(F) = \mathcal{N} \left( F \mid 0, \left( \begin{array}{c} K \\ v \end{array} \right) \right)$$  \hspace{1cm} (12)

where $K = (k(x_i, x_j))_{i,j}$ and $k(x) = (k(x, x_i))_i$. We then can evaluate the conditional distribution $p(f(x) \mid v_1, \ldots, v_k)$ (which is also a Gaussian) and sample from it to create a noisy version of $f(x)$. We repeat this procedure whenever we need to access to the value of one of the loss in $F_{\text{train}}$ at a point $x$.

The following lists details how $\mathcal{X}$ and $\mathcal{V}$ were sampled for each modalities of $F_{\text{train}}$.

- **Valleys**: We set $\mathcal{X} = \{0_{2^2}\}$ and sample $v_1$ uniformly at random in $[-5, 0]$. We sample one value $\lambda_1$ in a positive truncated Gaussian distribution centered at 10 with variance 2. We then multiply it by a ratio $\rho$ uniformly sampled at random inside the interval $[100, 200]$ to obtain $\lambda_2 = \rho \lambda_1$. We sample uniformly at random $\phi$ in $[0, 2\pi]$, and create $S = R_{\phi}^T \text{diag}(\lambda_1, \lambda_2) R_{\phi}$ where $R_{\phi}$ is the two-dimensional rotation matrix of angle $\phi$. By that mean, we are able to create valleys of different width and orientation.

- **Saddles** : we sample 4 points $x_1, \ldots, x_4$ uniformly at random within each quarter of the square $[-1, 1]^2$ and place them in $\mathcal{X}$. We assign a random value to each one (sampled from a Gaussian distribution) so that to opposite points have values of similar signs. We then sample $\lambda_1, \lambda_2$ in a truncated positive Gaussian distribution centered at 10 and with variance 2. We sample a random angle $\phi$ and compute $S = R_{\phi}^T \text{diag}(\lambda_1, \lambda_2) R_{\phi}$.

- **Plateau+cliffs**: $\mathcal{X} = \{0_{2^2}\}$ and generate a single value $v_1$ from a positive truncated Gaussian distribution centered in -5 with variance 2. We then create a matrix $S$ in the same fashion as for the previously described landscape.

### 6.2 Precisions on $F_{\text{test}}$

We provide here some contour plots for the functions used in the meta-test dataset. We also provide their mathematical expression, as well as the position of their global optimum and the position of the initial iterate we used in our experiment.

#### 6.2.1 Rosenbrock’s function

Rosenbrock’s function’s analytical expression is:

$$f(x) = 100(x_1 - x_0^2)^2 + (x_0 - 1)^2$$  \hspace{1cm} (13)

and its contour plot is shown in 8.

#### 6.2.2 Ackley’s function

Aclely’s function’s analytical expression is:

$$f(x) = -20 \exp \left( -0.2 \sqrt{\frac{1}{2} (x_1^2 + x_2^2)} \right) - \exp \left( \frac{1}{2} \cos 2\pi x_1 + \frac{1}{2} \cos 2\pi x_2 \right) + 20 + e^1$$  \hspace{1cm} (14)

and its contour plot is shown in 9.
6.2.3 Rastrigin’s function

Rastrigin’s function’s analytical expression is:

\[ f(x) = 20 + \sum_{i=1}^{2} (x_i^2 - 10 \cos(2\pi x_i)) \]  

and its contour plot is shown in 10.
6.2.4 Maccornick’s function
Maccornick’s function’s analytical expression is:

\[ f(x) = \sin(x_1 + x_2) + (x_1 - x_2)^2 - 1.5x_1 + 2.5x_2 + 1 \]  \hspace{1cm} (16)

and its contour plot is shown in 11.

Figure 11: Contour plot of the Maccornick function. The black circle indicates the position of the optimum and the blue square the position of the initial iterate.

6.2.5 Styblinski’s function
Styblinski’s function’s analytical expression is:

\[ f(x) = \frac{1}{2} \sum_{i=1}^{2} (x_i^4 - 16x_i^2 + 5x_i) \]  \hspace{1cm} (17)

and its contour plot is shown in 12.

Figure 12: Contour plot of the Styblinski function. The black circle indicates the position of the optimum and the blue square the position of the initial iterate.

6.2.6 Beale’s function
Beale’s function’s analytical expression is:

\[ f(x) = (1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_1x_2^2)^2 + (2.625 - x_1 + x_1x_2^3)^2 \]  \hspace{1cm} (18)

and its contour plot is shown in 13.
Figure 13: Contour plot of the Beale function. The black circle indicates the position of the optimum and the blue square the position of the initial iterate.