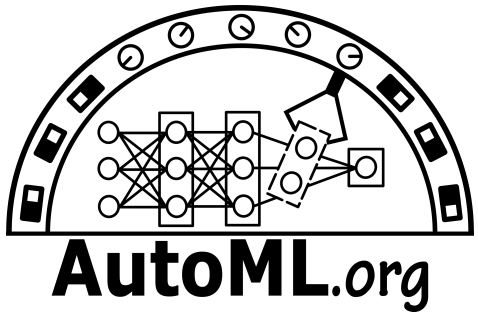
Automated Machine Learning
(AutoML)

M. Lindauer F. Hutter

University of Freiburg



Lectures 9+10:
Meta-Learning I+II

Where are we? The big picture

- Introduction
- Background
  - Design spaces in ML
  - Evaluation and visualization
- Hyperparameter optimization (HPO)
  - Bayesian optimization
  - Other black-box techniques
  - More details on Gaussian processes
- Pentecost (Holiday) – no lecture
- Architecture search I + II

→ Meta-Learning I + II

- Beyond AutoML: algorithm configuration and control
- Project announcement and closing

Learning Goals

After these two lectures, you will be able to . . .

- explain the high-level principles of meta-learning and its facets
- use algorithm selection to predict the best algorithm for a dataset
- improve the performance of AutoML tools by warmstarting its search
- transfer knowledge between DNNs by determining initial weights
- learn a DNN which can train another DNNs
- learn a policy to optimize a black-box function
- learn policies to either replace algorithm components or to control hyperparameters

Warning: I gave my best to unify the notation of different papers here. Please keep that in mind if you read the original papers which use different notations.

Further Material

There were two great tutorials on meta-learning recently:

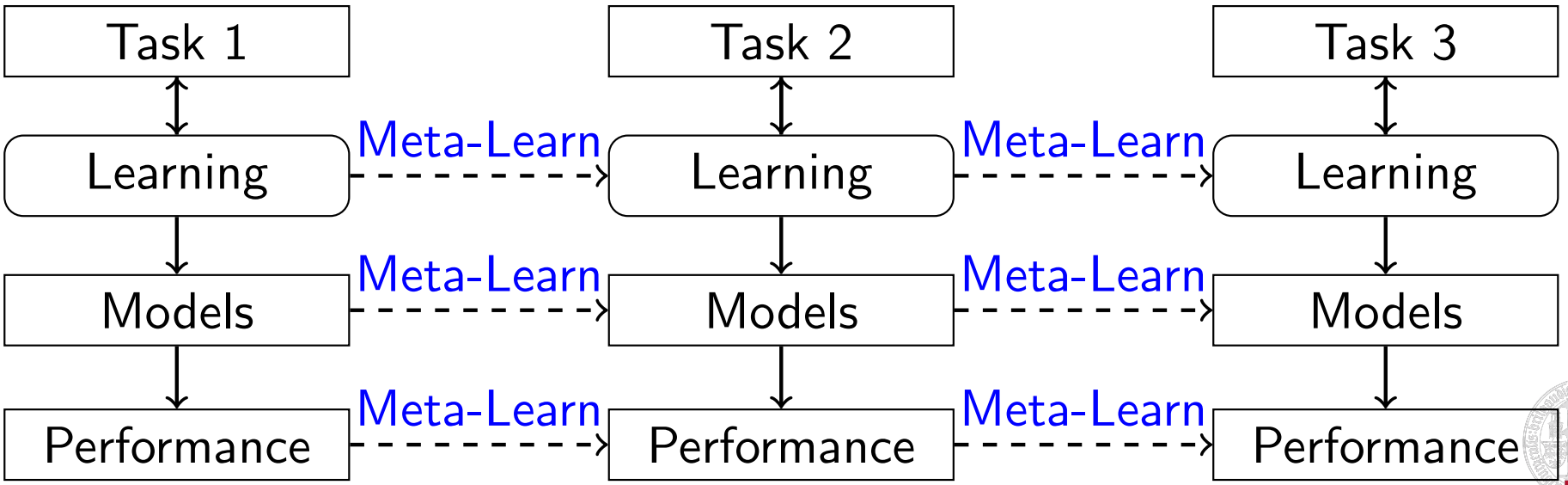
- NeurIPS'18: Frank Hutter and Joaquin Vanschoren "Automatic Machine Learning (AutoML): A Tutorial"
  `https://videoken.com/embed/5A4xbv5nd8c`
- ICML'19: Chelsea Finn and Sergey Levine on
  "Meta-Learning: from Few-Short Learning to Rapid Reinforcement Learning"
  `https://www.facebook.com/icml.imls/videos/400619163874853/`

- We can't cover all the stuff in details
- Strong recommendation to watch both tutorials
- Parts of the material used today was inspired by these tutorials

The Idea of Meta-Learning

- Learning essentially never stops
  - We learn several models on the same task (e.g., dataset)
  - We learn models on new tasks
- Learning is often done from scratch

⇝ We humans don't start from scratch all the time, but we learned how to learn!



Example for Human Meta-Learning

Braque

Cezanne

Who painted that?

Most likely most of you can identify the painter correctly, although I presented only three pictures of each.

Recap Supervised Learning

Dataset:

$$\mathcal{D} = \{(x_1, y_1), \ldots, (x_k, y_k)\} \tag{1}$$

Learning a model ϕ (e.g., weights of a neural network):

$$\arg\max_{\phi} \log p(\phi|\mathcal{D}) \tag{2}$$

$$= \arg\max_{\phi} \log p(\mathcal{D}|\phi) + \log p(\phi) \tag{3}$$

$$= \arg\max_{\phi} \sum_i \log p(y_i|x_i, \phi) + \log p(\phi) \tag{4}$$

Challenge:

- Learning starts from scratch
  (e.g., we initially have no clue how to learn a good ϕ)
- we might only have very few examples in $\mathcal{D}$ (i.e., k is small)

The Meta Learning Problem

Dataset:

$$\mathcal{D} = \{(x_1, y_1), \ldots, (x_k, y_k)\} \tag{5}$$

Set of datasets (meta-datasets):

$$\mathcal{D}_{\text{meta}} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n, \} \tag{6}$$

Can we include these meta-datasets to improve learning on $\mathcal{D}$?

$$\arg\max_{\phi} \log p(\phi | \mathcal{D}, \mathcal{D}_{\text{meta}}) \tag{7}$$

Idea: Instead of keeping $\mathcal{D}_{\text{meta}}$ forever, we want to distill the knowledge into meta-parameters θ: $p(\theta | \mathcal{D}_{\text{meta}})$

The Meta Learning Problem

In meta-learning, we want to learn:

$$\arg\max_{\phi} \log p(\phi|\mathcal{D}, \mathcal{D}_{\text{meta}}) \tag{8}$$

$$= \arg\max_{\phi} \log \int_{\Theta} p(\phi \mid \mathcal{D}, \theta) p(\theta \mid \mathcal{D}_{\text{meta}}) d\theta \tag{9}$$

$$\approx \arg\max_{\phi} \log p(\phi|\mathcal{D}, \theta^*) + \log p(\theta^*|\mathcal{D}_{\text{meta}}) \tag{10}$$

$$= \arg\max_{\phi} \log p(\phi|\mathcal{D}, \theta^*) \tag{11}$$

The meta-learning problem is:

$$\theta^* \in \arg\max_{\theta} \log p(\theta|\mathcal{D}_{\text{meta}}) \tag{12}$$

AutoML $\subset$ Meta-Learning

- AutoML can be seen as a special case of meta-learning
- θ could be:
  - a hyperparameter configuration (λ)
  - a neural network architecture
- What would be $\mathcal{D}_{\text{meta}}$ here? 
  - the train and validation dataset?
  - a dataset on which we optimized λ (e.g., CIFAR-10) such that we can use it on another dataset (e.g., imagenet)

Meta-Learning $\subset$ AutoML

- Meta-learning can be powerful to complement AutoML
- We can learn a lot of things $\mathcal{D}_{\text{meta}}$ to improve the performance on new datasets
  - pre-initialization of networks weights
    (e.g., pre-train on imagenet and fine-tune on a new dataset)
  - learning a meta-DNN to predict how to train another target-DNN
    (more next week)
- Open question: If we can learn how to learn,
  do we need AutoML anymore?

Idea: Algorithm Selection

- By applying ML to many different datasets,
  humans get an intuition what works well on which dataset
- E.g., humans came up with rule of thumbs:
  "On small datasets we use an SVM, on mid-size dataset a RF
  and on big data a DNN"
- Although these rule thumbs are quite useful, they are often not
  correct and simplify real applications too strongly
- Question: Can we learn from data which algorithm we should use for
  a given dataset?

Definition: Algorithm Selection [Rice 1976]

Definition

Given

- a distribution $\mathcal{D}_{\text{meta}}$ of meta datasets,
- a portfolio of algorithms $\mathcal{A} \in \mathcal{P}$,
- and a cost metric $c : \mathcal{P} \times \mathcal{D}_{\text{meta}} \to \mathbb{R}$,

the *per-instance algorithm selection problem* is to find a mapping $s : \mathcal{D} \mapsto \mathcal{A}$ that optimizes

$$\arg\min_{s} \int_{\mathcal{D}_{\text{meta}}} c(s(\mathcal{D}), \mathcal{D}) p(\mathcal{D}) d\mathcal{D}$$

Meta-Features in Machine Learning

To learn the mapping $s : \mathcal{D} \mapsto \mathcal{A}$, we need a way to represent $\mathcal{D}$

Features

Observations

f_1	f_2	f_3	f_4	f_5	f_6	y
1	31	22	0	23	1	1
2	21	22	0	32	1	1
4	32	11	1	1	1	0

Labels

What kind of "Meta-features" could describe a dataset?

- number of observations (samples)
- number of features
- number of classes
- number of categorical features
- number of numeric features
- percentage of numeric features
- number of missing features
- Probing: accuracy of a small decision tree

Assumptions for Traditional Algorithm Selection

We assume for a discrete set of datasets from $\mathcal{D}_{\text{meta}}$ that

- we have pre-computed meta-features on all datasets
  - if our cost metric is not related to runtime,
    we assume that the cost for computing the meta features is
    negligible compared to training a model
- we have measured the cost $c(\mathcal{A}, \mathcal{D})$ on all possible pairs $\langle \mathcal{A}, \mathcal{D} \rangle$

$\rightsquigarrow$ We have two matrices as training data,
one with meta-features and one with cost values

Algorithm Selection as a Regression Problem

- Given a new dataset and its meta-features $f(\mathcal{D})$
- we "only" need to predict the performance of all algorithms $\mathcal{A} \in \mathcal{P}$
- For each algorithm $\mathcal{A} \in \mathcal{P}$, we train a regression model to predict its cost $c(\mathcal{A}, \mathcal{D})$ given $\mathcal{D}$'s meta features:

$$\hat{m}_{\mathcal{A}} : f(\mathcal{D}) \mapsto c(\mathcal{A}, \mathcal{D})$$

- Selection for given test $\mathcal{D}_{\text{test}}$:

$$\arg\min_{\mathcal{A} \in \mathcal{P}} \hat{m}_{\mathcal{A}}(f(\mathcal{D}_{\text{test}}))$$

- Remarks:
  - As regression approach, we can essentially use whatever you want e.g., ridge regression, a random forest or a neural network
  - We could also train one joint model with indicator features for the algorithms

Algorithm Selection as a Classification Problem

- Learning a regression problem is an indirect approach solving the algorithm selection problem
- Can we directly solve the multi-class classification problem?

$$s : \mathcal{D} \mapsto \mathcal{A}$$

- We can use common approach for multi-class classification
- Algorithm selection has a special characteristic:
  Some datasets are more sensitive to algorithm selection than others
  - for very easy datasets, it might not matter which algorithm to use
  - for very hard datasets (e.g., because of poor data quality), all algorithms might fail
  - for all others, it might be important to select the right algorithm

Pairwise Cost-Sensitive Classification Problem [Xu et al. 2011]

- The cost-sensitivity of a dataset $\mathcal{D}$ to algorithm selection across a portfolio is somehow hard to measure
- For each pair of algorithms $\langle \mathcal{A}_i, \mathcal{A}_j \rangle$ we can easily measure the cost-sensitivity by

$$sim(\mathcal{D}, \mathcal{A}_i, \mathcal{A}_j) = ||c(\mathcal{A}_i, \mathcal{D}) - c(\mathcal{A}_j, \mathcal{D})||$$

- Train for each pair of algorithms a classification model $s_{\mathcal{A}_i, \mathcal{A}_j}$ predicting which algorithm performs better
- consider for each training dataset $\mathcal{D}_{\text{train}}$ the $sim(\mathcal{D}_{\text{train}}, \mathcal{A}_i, \mathcal{A}_j)$
  - cost-sensitive classification is often possible by modifying the loss function or split metric in tree-based approaches
- Selection for given test $\mathcal{D}_{\text{test}}$:

$$\arg\max_{\mathcal{A} \in \mathcal{P}} \sum_{\mathcal{A}' \in \mathcal{P}} s_{\mathcal{A}, \mathcal{A}'}(f(\mathcal{D}_{\text{test}}))$$

Remarks on Algorithm Selection

- Pairwise cost-sensitive classification requires $\sim |\mathcal{P}|^2/2$ models
- From our experience, pairwise cost-sensitive approaches often perform better than regression approaches, but not always
- We use machine learning to optimize machine learning
  - Also the meta learning level can benefit from AutoML [Lindauer et al. 2015]
- Algorithm selection is not limited to machine learning
- Algorithm selection can be applied to any application where
  1. we have a discrete portfolio of algorithms to choose from
  2. the tasks (a.k.a. datasets) are sensitive to the algorithm choice (so called heterogeneous tasks/instances)
  3. we have informative meta-features (a.k.a. instance features) (only partially true for ML)
- There is a benchmark library for algorithm selection data called ASlib (`www.aslib.net`) [Bischl et al. 2016]

Warmstarting

- Recap: Instead of starting from a random configuration we often start from a expert-defined configuration for hyperparameter optimization (HPO)
- We also know that the default configuration often does not perform well on a new dataset
  - Otherwise there would be no point in HPO
- Can we learn from previous datasets $\mathcal{D}_{\text{meta}}$ how to initialize HPO? (i.e., running an initial design)
  - the same ideas also apply to NAS
  - for simplicity we focus on HPO

Multi-Configuration Initialization

- Idea: Instead of a single starting point, use a portfolio Λ_{init} as an initial design
- Assume that we applied HPO already to many datasets $\mathcal{D}_{\text{meta}}$ and obtained a well-performing configuration on all of them $\hat{\lambda}_{\mathcal{D}}$
- Straightforward idea:

$$\Lambda_{init} = \bigcup_{\mathcal{D} \in \mathcal{D}_{\text{meta}}} \{\hat{\lambda}_{\mathcal{D}}\}$$

- Problems:  ?
  - If $|\mathcal{D}_{\text{meta}}|$ is too large, HPO will be dominated by Λ_{init}
  - Λ_{init} has potentially a lot of similar configurations
    $\rightsquigarrow$ Λ_{init} is not complementary, thus inefficient for warmstarting

Portfolio Construction Approach

- Given
  - a large portfolio of configurations
  - the cost of each configuration on all $\mathcal{D} \in \mathcal{D}_{\text{meta}}$
- Goal: Select a subset of complementary configurations that cover $\mathcal{D}_{\text{meta}}$ well

$$\begin{aligned} &\arg\max_{\Lambda' \subset \Lambda_{init}} c(\Lambda') \\ &\text{s.t. } |\Lambda'| \leq k \end{aligned}$$

- where k is a user-defined upper bound on the portfolio size

$$c(\Lambda') = \frac{1}{|\mathcal{D}_{\text{meta}}|} \sum_{\mathcal{D} \in \mathcal{D}_{\text{meta}}} \min_{\lambda \in \Lambda'} c(\lambda, \mathcal{D})$$

- Even if all cost values are known, exhaustive search is infeasible for reasonable sizes of Λ_{init} and k

Greedy Portfolio Construction Approach

- We can greedily construct the portfolio by adding one configuration at a time
- In each iteration, we have already a selected portfolio Λ_i and want to add a single configuration maximizing its contribution

$$\lambda_{i+1} \in \arg\max_{\lambda \in \Lambda} c(\Lambda_i) - c(\Lambda_i \cup \{\lambda\}) \tag{13}$$

- Re-iterate: $\Lambda_{i+1} = \Lambda_i \cup \{\lambda_{i+1}\}$
- Remark: $c(\Lambda)$ is defined as a submodular function
  (i.e., adding a specific λ early in the process gains you more than adding it in later iterations)

$\rightsquigarrow$ At most away from optimum by factor of 0.63
(see [Streeter & Golovin '07])

Feature-based Initial Portfolios

- Instead of throwing away parts of Λ_{init},
  we can try to select on the fly which ones to use
- Idea similar to algorithm selection
  1. Compute meta-featues for a given new dataset
  2. Select k-nearest datasets from $\mathcal{D}_{\text{meta}}$ wrt meta-features
  3. Use the optimized λ of these k datasets

- Problem: If you have very little time for AutoML, you don't want to invest time in meta-feature computation

Model-Warmstarting

- Many HPO optimizers make use of some kind of a predictive model, e.g., Bayesian optimization
- By running HPO over and over again on different datasets, we can actually learn something about the search landscape
  - E.g., what are bad regions of the configuration space in general
- Given: n predictive models $\hat{f}_{\mathcal{D}_i} : \mathbf{\Lambda} \rightarrow \mathbb{R}$ from HPO on $\mathcal{D}_{\text{meta}}$
- How can we use these $\hat{f}_{\mathcal{D}_i}$ to speed up HPO?

Model-Warmstarting: Direct Predictions

- You could train a model $\hat{f}_{\mathcal{D}_{\text{meta}}} : \lambda \times \mathcal{D} \mapsto c(\lambda, \mathcal{D})$ to predict the cost of a configuration λ on a dataset $\mathcal{D}$
- Use $\hat{f}_{\mathcal{D}_{\text{meta}}}$ to augment your search (e.g., in BO) and update it with newly collected data in the current dataset
- Problems:
  - Updating this big model $\hat{f}_{\mathcal{D}_{\text{meta}}}$ might take too long
  - You can't completely ignore misleading knowledge from dissimilar datasets

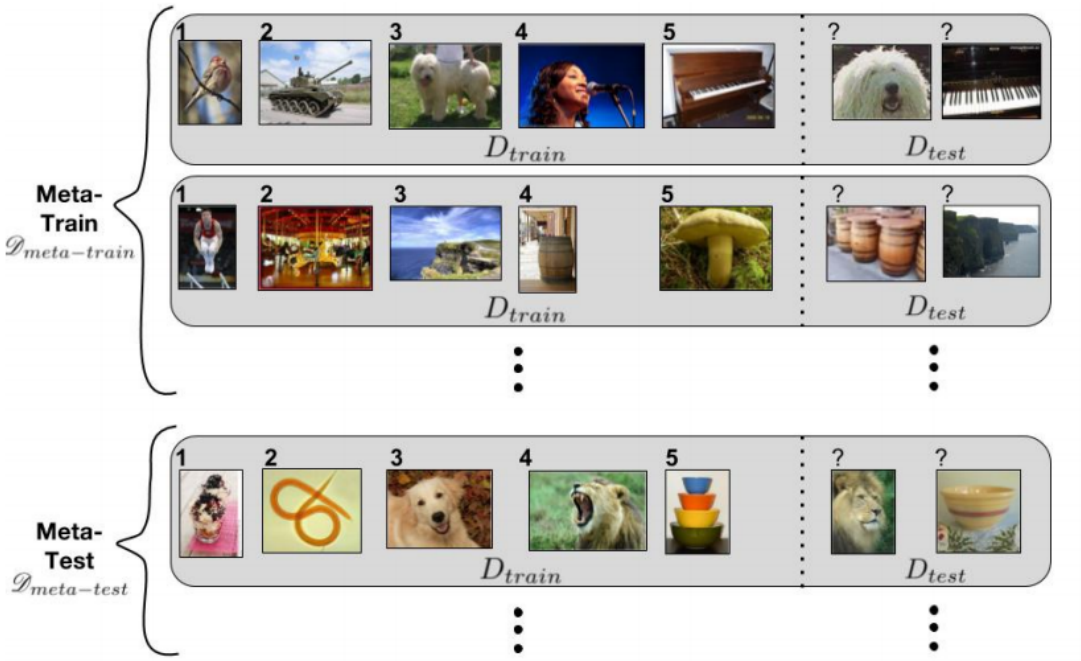
Model-Warmstarting: Combined Models

- Instead of one big model, you can train an ensemble of models, e.g., a linear combination

$$\hat{f}_{\texttt{combined}} = w_0 \hat{f} + \sum_{\mathcal{D}_i \in \mathcal{D}_{\text{meta}}} w_i \hat{f}_{\mathcal{D}_i}$$

- Advantages:
  - only $\hat{f}$ and $\mathbf{w}$ has to be updated for new observations
  - after "some" observations, it will start to focus on $\hat{f}$
  - Thus it can recover from misleading knowledge
- We can use SGD to train weights $\mathbf{w}$ on our recent observations
- Possible to also use more complex models such as DNNs
- a learnt initial design and model-warmstarting can be combined
  - combination can perform even better than the better of these two

DataSets: Meta-Train, Meta-Test



[Ravi and Larochelle. 2017]

Notation:

- meta-train: $\mathcal{D}_{\text{meta}} = \{(\mathcal{D}_1^{\text{train}}, \mathcal{D}_1^{\text{test}}), \ldots, \mathcal{D}_n^{\text{train}}, \mathcal{D}_n^{\text{test}})\}$

$\rightsquigarrow$ Assumption: At meta-test we only have a few examples ("few shots") from a similar task

Optimization-based Initialization [Fin et al. 2017]

- Idea: Acquire ϕ through optimization

$$\arg\max_{\phi_i} \log p(\mathcal{D}_i^{\text{train}}|\phi) + \log p(\phi_i|\theta)$$

$\rightsquigarrow$ Meta-parameters θ serve as a prior

- One successful form of prior knowledge is the initialization of DNNs for fine-tuning (at test time):

$$\phi \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}^{\text{train}})$$

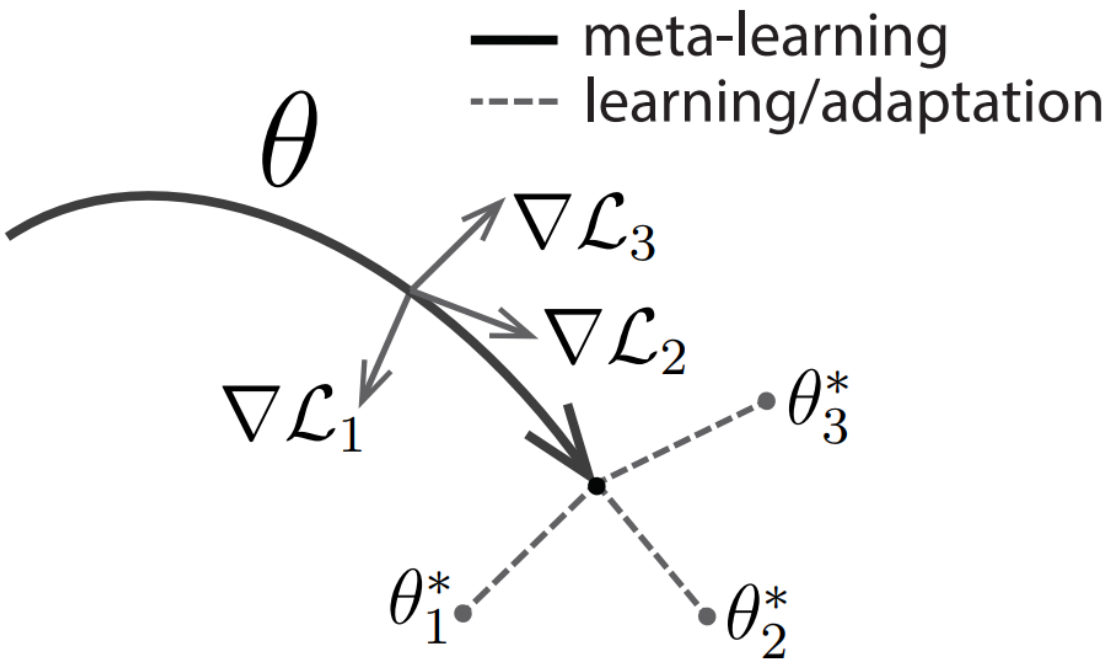
where θ are pre-trained weights and $\mathcal{D}^{\text{train}}$ is data for a new task

- Meta-learning task:

$$\arg\min_\theta \sum_{\text{task } i} \mathcal{L}(\theta - \alpha \nabla_\theta \mathcal{L}(\theta, \mathcal{D}_i^{\text{train}}), \mathcal{D}_i^{\text{test}})$$

Model-Agnostic Meta-Learning (MAML) [Fin et al. 2017]

$$\arg\min_{\theta} \sum_{\text{task } i} \mathcal{L}(\theta - \alpha \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_i^{\text{train}}), \mathcal{D}_i^{\text{test}})$$



where θ_i^* are the optimal weights for Task i

MAML Algorithm [Fin et al. 2017]

Input : Distribution $p(\mathcal{T})$ over tasks, learning rates α and β

randomly initilize θ;
while *not done* **do**
    Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$;
    foreach $\mathcal{T}_i$ **do**
        Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i})$ with respect to K examples;
        Compute adapted parameters with gradient descent: $\theta_i' \rightarrow \theta_i - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i})$;
    Update $\theta \rightarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T} \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$;
return *Pre-trained weights* θ

MAML Algorithm for Supervised Learning[Fin et al. 2017]

Input : Distribution $p(\mathcal{T})$ over tasks, learning rates α and β

randomly initilize θ;
while *not done* **do**
  Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$;
  foreach $\mathcal{T}_i$ **do**
    Sample K datapoints $\mathcal{D} = \{x^{(j)}, y^{(j)}\}$ from $\mathcal{T}_i$;
    Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i})$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$;
    Compute adapted parameters with gradient descent: $\theta_i' \rightarrow \theta_i - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i})$;
    Sample datapoints $\mathcal{D}' = \{x^{(j)}, y^{(j)}\}$ from $\mathcal{T}_i$ for the meta-update;
  Update $\theta \rightarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T} \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}'$ and $\mathcal{L}_{\mathcal{T}_i}$;
return *Pre-trained weights* θ

Discussion of MAML

- MAML can improve the performance on few-short learning task (i.e., we only have few examples for each task) where we use only a few update steps
- MAML uses second-order derivatives which makes it only applicable to small datasets
  - first-order approximation of MAML can also perform quite well
- MAML-fine-tuning requires backpropagation through the entire network
  - in "traditional" fine-tuning, we only fine-tune the last layer(s)

Learning to learn by gradient descent by gradient descent
by Andrychowicz et al. '16

Learning to Learn

Idea

- Learn algorithms directly, e.g., how to search in the weight space
- First idea: learn weight updates of a neural network

Learning to learn by gradient descent by gradient descent

[Andrychowicz et al'16]

Weight updates (note: θ denote DNN weights!):

$$\theta_{t+1} = \theta_t - \alpha_t \nabla f(\theta_t)$$

Even more general:

$$\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi)$$

where g is the optimizer and ϕ are the parameters of the optimizer g.
$\rightsquigarrow$ Goal: Optimize f wrt θ by learning g (resp. ϕ)

Learning to Learn: Objective [Andrychowicz et al'16]

$$\mathcal{L}(\phi) = \mathbb{E}\left[f(\theta^*(f, \phi))\right]$$

where $\mathcal{L}$ is a loss function and $\theta^*(f, \phi)$ are the optimized weights θ^* by using the optimizer parameterized with ϕ on function f.

$$\mathcal{L}(\phi) = \mathbb{E}\left[\sum_{t=1}^{T} w_t f(\theta_t)\right]$$

where w_t are arbitrary weights associated with each time step and

$$\theta_{t+1} = \theta_t + g_t$$
$$\begin{pmatrix} g_t \\ h_{t+1} \end{pmatrix} = m(\nabla_\theta f(\theta_t), h_t, \phi)$$

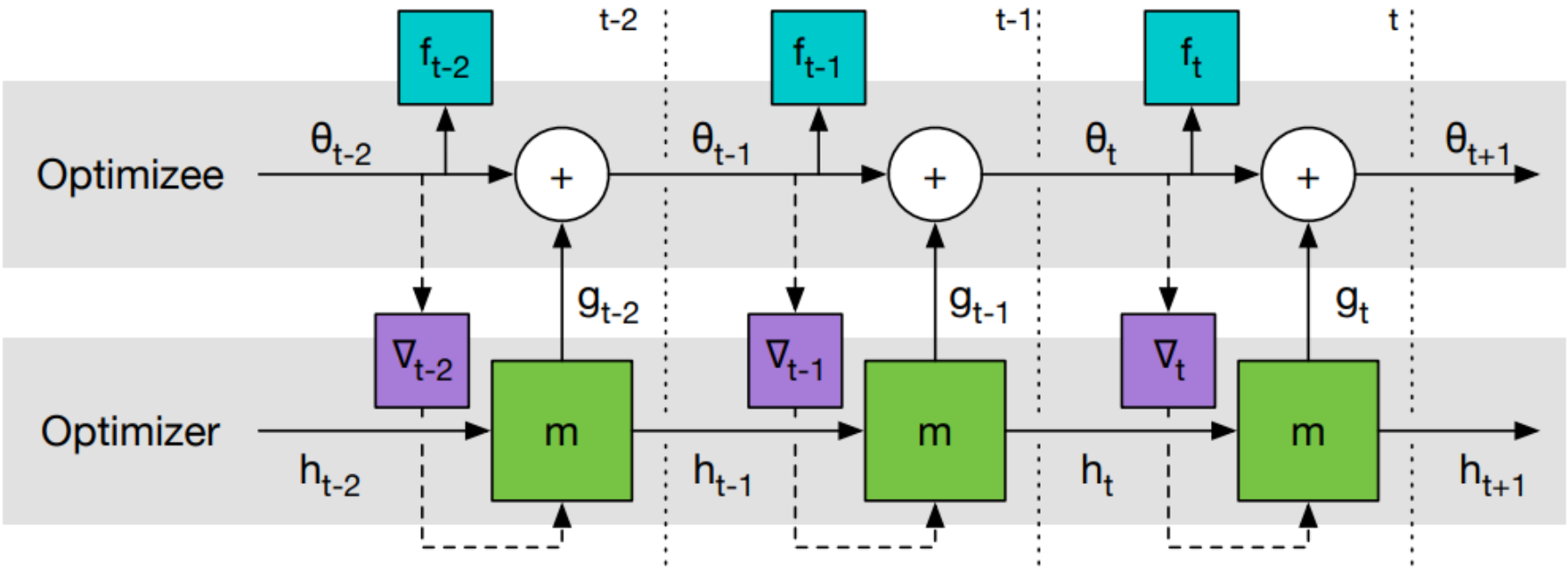
$\rightsquigarrow$ Goal: Learn m via ϕ by using gradient descent by optimizing $\mathcal{L}$

$\rightsquigarrow$ "Learning to learn gradient descent by gradient descent"

Learning to Learn: LSTM approach [Andrychowicz et al'16]

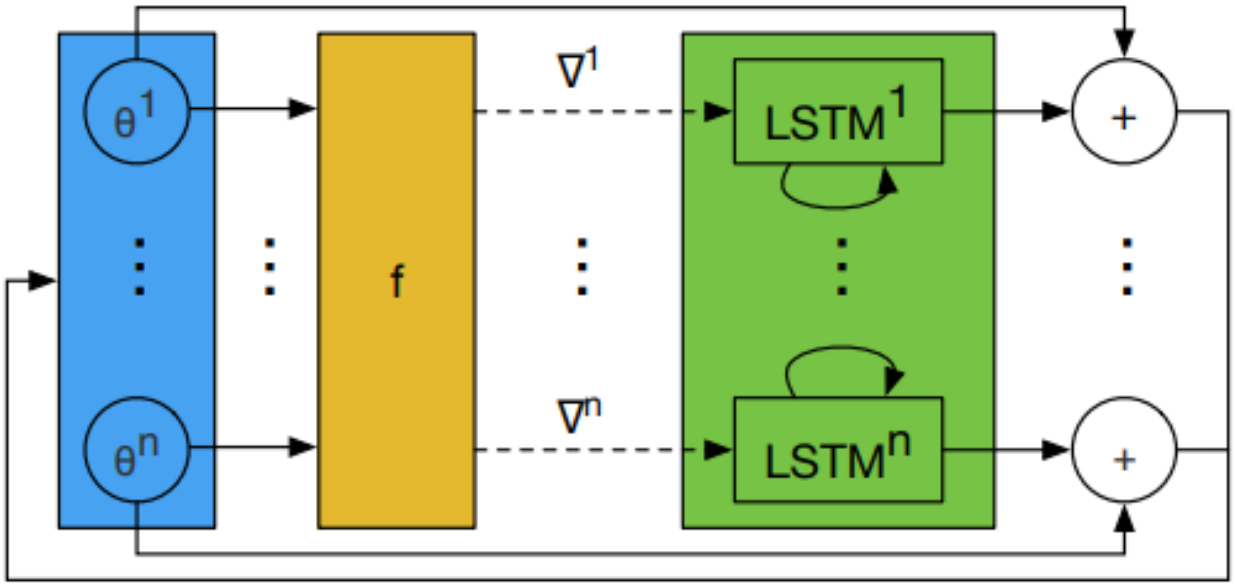
Optimizee Target network to be trained

Optimizer LSTM with hidden state h_t that predicts weight updates g_t



Learning to Learn: Coordinatewise LSTM optimizer

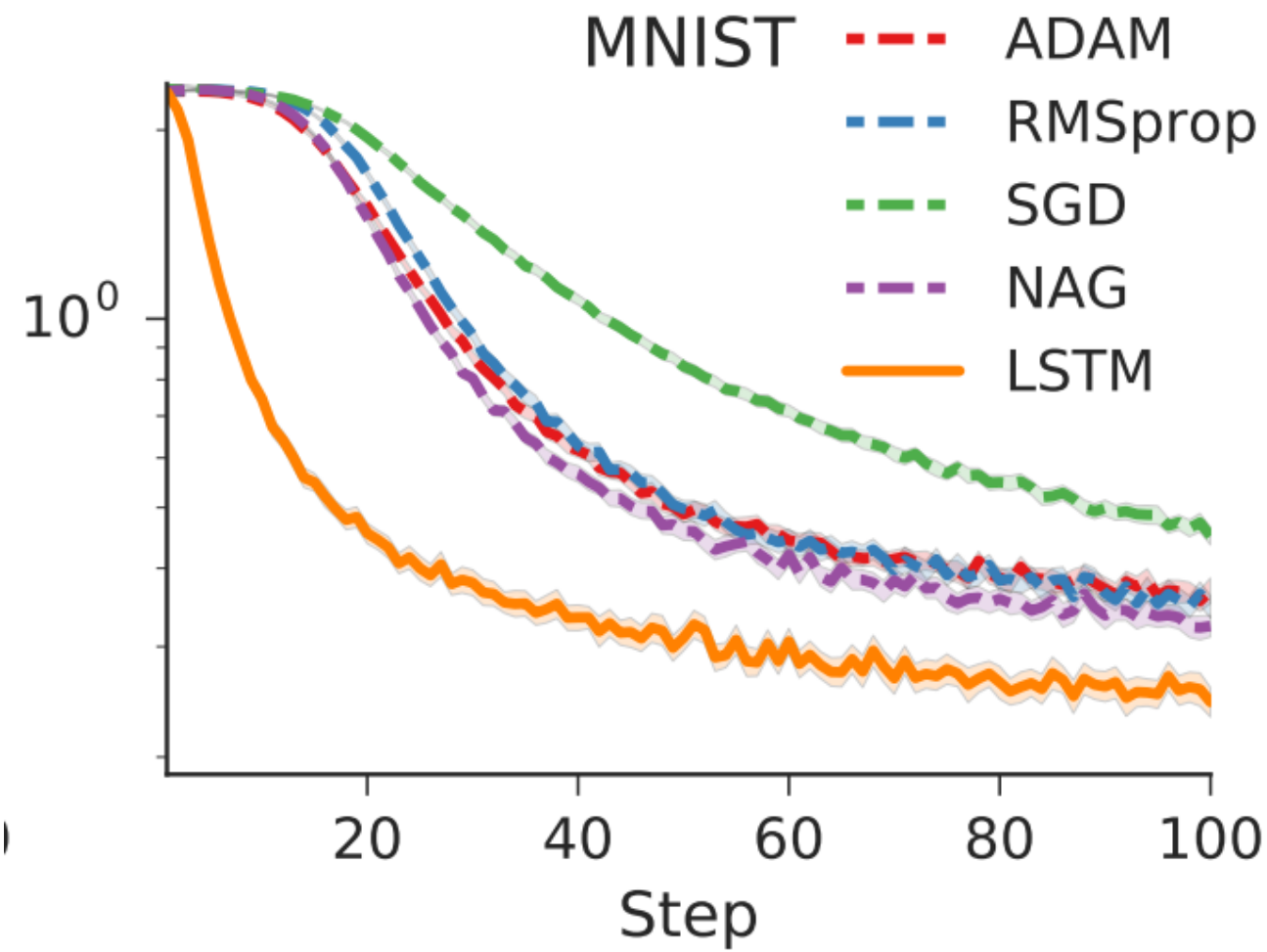
[Andrychowicz et al'16]



- One LSTM for each coordinate (i.e., weight)
- All LSTMs have shared parameters ϕ
- Each coordinate has its own separate hidden state

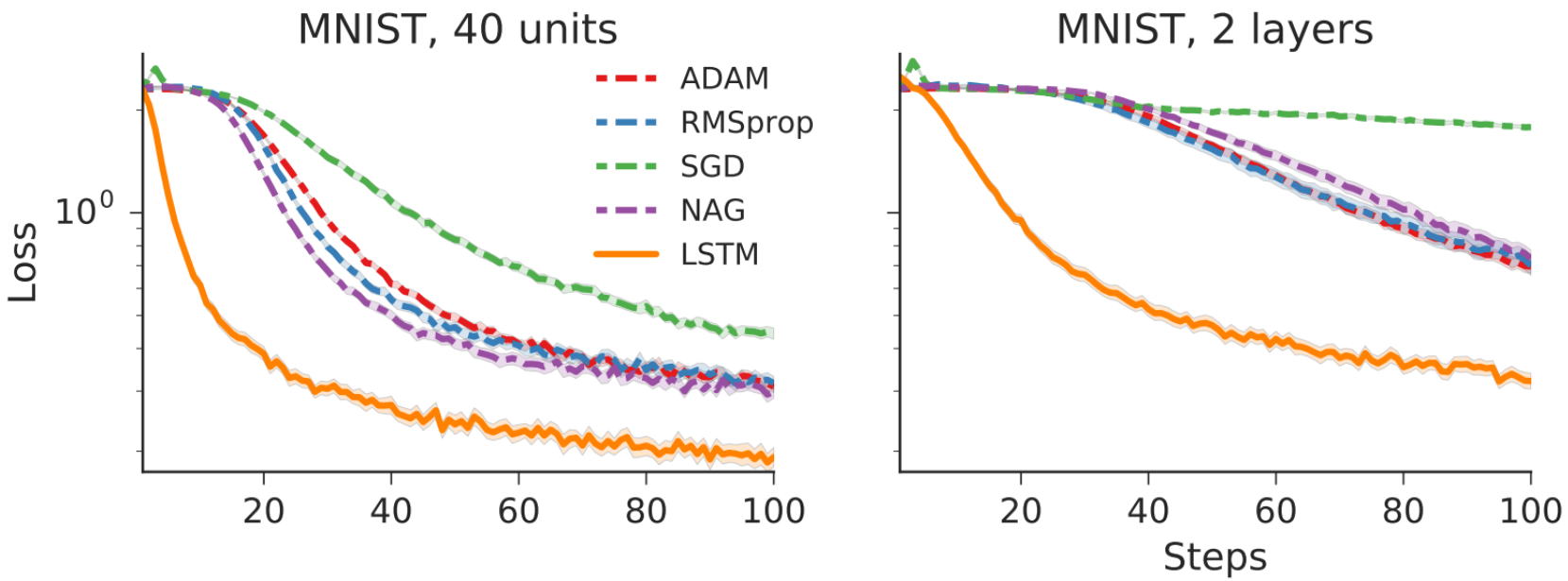
$\rightsquigarrow$ We can train the LSTM on k weights and apply it larger DNNs with k' weights, where $k \leq k'$

Learning to Learn with LSTM: Results [Andrychowicz et al'16]



Learning to Learn with LSTM: Results [Andrychowicz et al'16]

Changing the original architecture of the DNN:

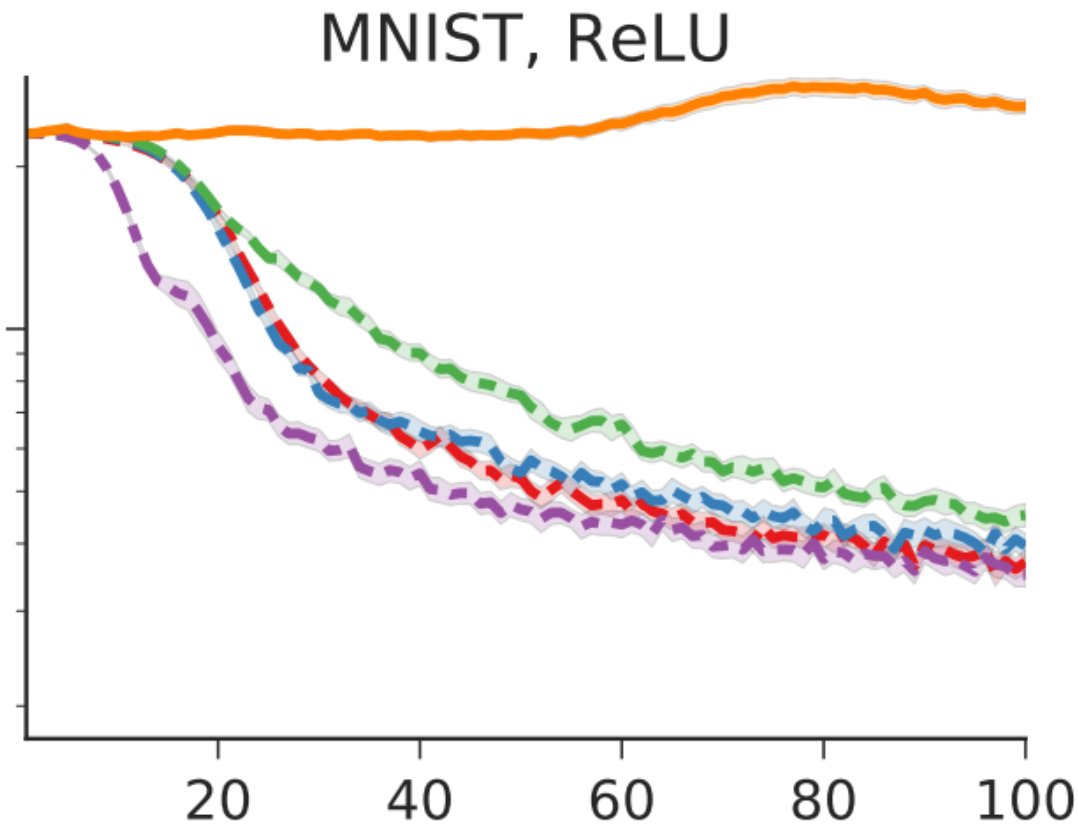


$\rightsquigarrow$ learnt optimizer is robust against some architectural changes

Learning to Learn with LSTM: Results

[Andrychowicz et al'16]

Changing the activation function to ReLU:



$\rightsquigarrow$ fails on other activation functions

Learning Step Size Controllers for Robust Neural Network Training by Daniel et al. '16

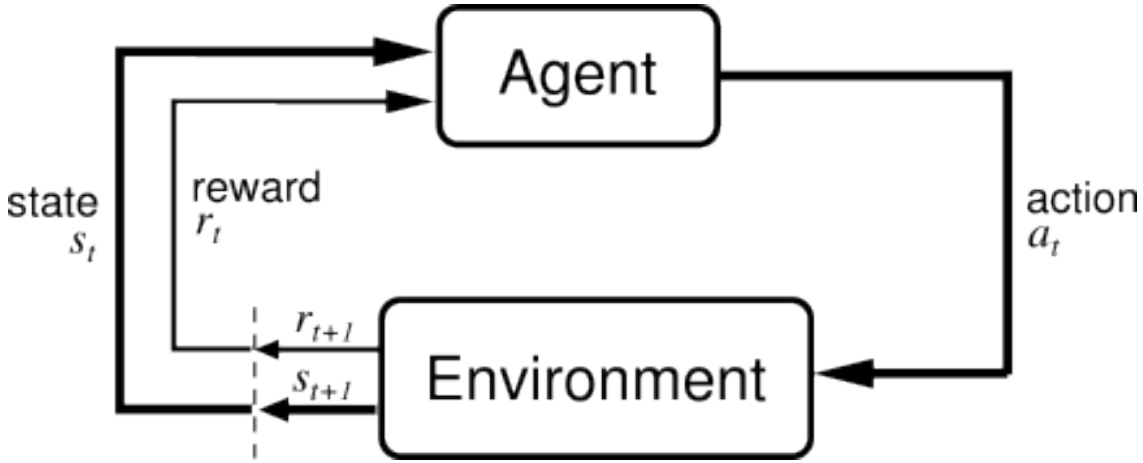
Learning Step Size Controllers [Daniel et al'16]

- Idea: Learn the hyperparameters of the weight update

$$\theta_{t+1} = \theta_t - \alpha_t \nabla f(\theta_t)$$

- For SGD, this would be for example the learning rate α
- Note: α have to be adapted in the course of the training
  - similar to learning rate schedules (e.g., cosine annealing)
- Note(ii): later we denote the learnt hyperparameters as λ
- Idea: Use reinforcement learning to learn a policy $\pi : s \mapsto a$ to control the learning rate (or other adaptive hyperparameters)

Recap: Reinforcement Learning [Barto & Sutton; RL Book]



$$\pi^* \in \arg\max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=1}^{T} \gamma^{t-1} r_t \mid s_0 \right]$$

The goal is to find the optimal policy $\pi^* : s \mapsto a$ starting in a state s_0 (or in expectation of s_0) s.t. following π^* from s maximizes the expected (discounted) reward R.

RL for Step Size Controllers: State [Daniel et al'16]

Predictive change in function value:

$$s_1 = \log\left(\mathsf{Var}(\Delta\tilde{f}_i)\right)$$

$$\Delta\tilde{f}_i = \tilde{f}(x_i; \theta + \delta\theta) - f(x_i; \theta)$$

where $\tilde{f}(x_i; \theta + \delta\theta)$ is done by a first order Taylor expansion

Disagreement of function values:

$$s_2 = \log\left(\mathsf{Var}(f(x_i; \theta))\right)$$

Discounted Average (smoothing noise from mini-batches):

$$\hat{s}_i \leftarrow \gamma\hat{s}_i + (1-\gamma)s_i$$

Uncertainty Estimate (noise level):

$$s_{K+i} \leftarrow \gamma s_{K+i} + (1-\gamma)(s_i - \hat{s}_i)^2$$

RL for Step Size Controllers: Learning [Daniel et al'16]

Reward (average loss improvement over time):

$$r = \frac{1}{T-1} \sum_{t=2}^{T} (\log(\mathcal{L}_{t-1}) - \log(\mathcal{L}_t))$$

Optimal Policy:

$$\pi^*(\lambda \mid s) \in \arg\max_{\pi} \int \int p(s)\pi(\lambda \mid s) r(\lambda, s) \mathtt{d}s\mathtt{d}\lambda$$

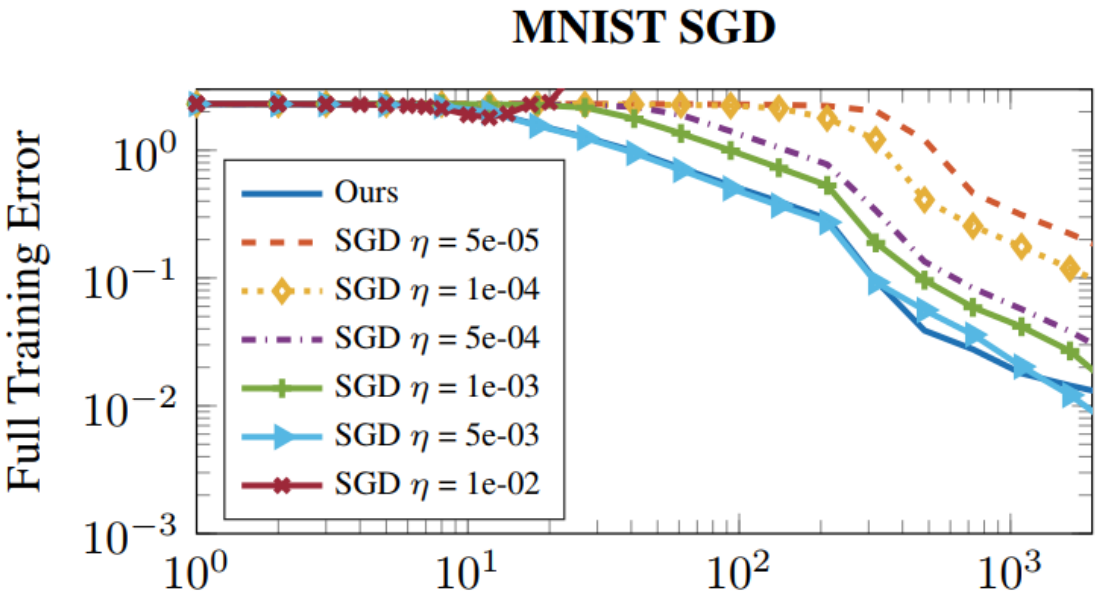
The policy π is learnt as a controller $\lambda = g(s, \phi)$; leading to

$$\pi^*(\phi) \in \arg\max_{\pi} \int \int p(s)\pi(\phi) r(g(s;\phi), s) \mathtt{d}s\mathtt{d}\lambda$$

- can be learnt for example via Relative Entropy Policy Search (REPS) [Peter et al. 2010]

RL for Step Size Controllers: Training [Daniel et al'16]

- Goal: obtain robust policies,
  i.e., good performance for many different DNN architectures
  - ⇝ Sample architectures e.g., with different numbers of filters and layers
  - ⇝ (Sub-)Sample dataset
  - ⇝ Sample number of optimization steps



"Ours" refers to the approach by [Daniel et al'16] and η is the learning rate

Learning to Learn without Gradient Descent by Gradient Descent by Chen et al'17

Learning Black-box Optimization [Chen et al'17]

Black Box Optimization Setting

1. Given the current state of knowledge h_t propose a query point x_t
2. Observe the response y_t
3. Update any internal statistics to produce h_{t+1}

Learning Black Box Optimization

Essentially, same idea as before:

$$h_t, x_t = \text{RNN}_\phi(h_{t-1}, x_{t-1}, y_{t-1})$$
$$y_t \sim p(y|x_t)$$

- Using recurrent neural network (RNN) to predict next x_t.
- h_t is the internal hidden state
- Remark: in a black-box setting, we don't have gradient information!

Learning Black-box Optimization: Loss Functions

[Chen et al'17]

- Sum loss: Provides more information than final loss

$$\mathcal{L}_{\text{sum}}(\phi) = \mathbb{E}_{f, y_{1:T-1}} \left[\sum_{t=1}^{T} f(x_t) \right]$$

- EI loss: Try to learn behavior of Bayesian optimizer based on expected improvement (EI)
  - requires model (e.g., GP)

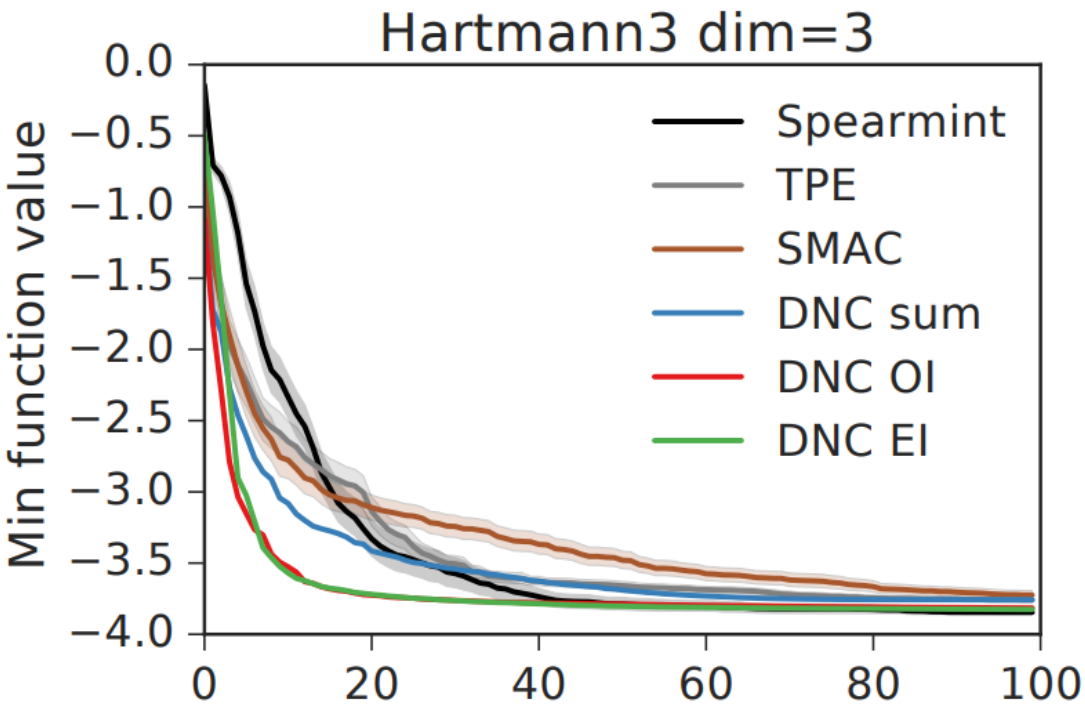
$$\mathcal{L}_{\text{EI}}(\phi) = -\mathbb{E}_{f, y_{1:T-1}} \left[\sum_{t=1}^{T} \text{EI}(x_t | y_{1:t-1}) \right]$$

- Observed Improvement Loss:

$$\mathcal{L}_{\text{OI}}(\phi) = \mathbb{E}_{f, y_{1:T-1}} \left[\sum_{t=1}^{T} \min \left\{ f(x_t) - \min_{i<t}(f(x_i)), 0 \right\} \right]$$

Learning Black-box Optimization: Results

[Chen et al'17]



- Hartmann3 is an artifical function with 3 dimensions

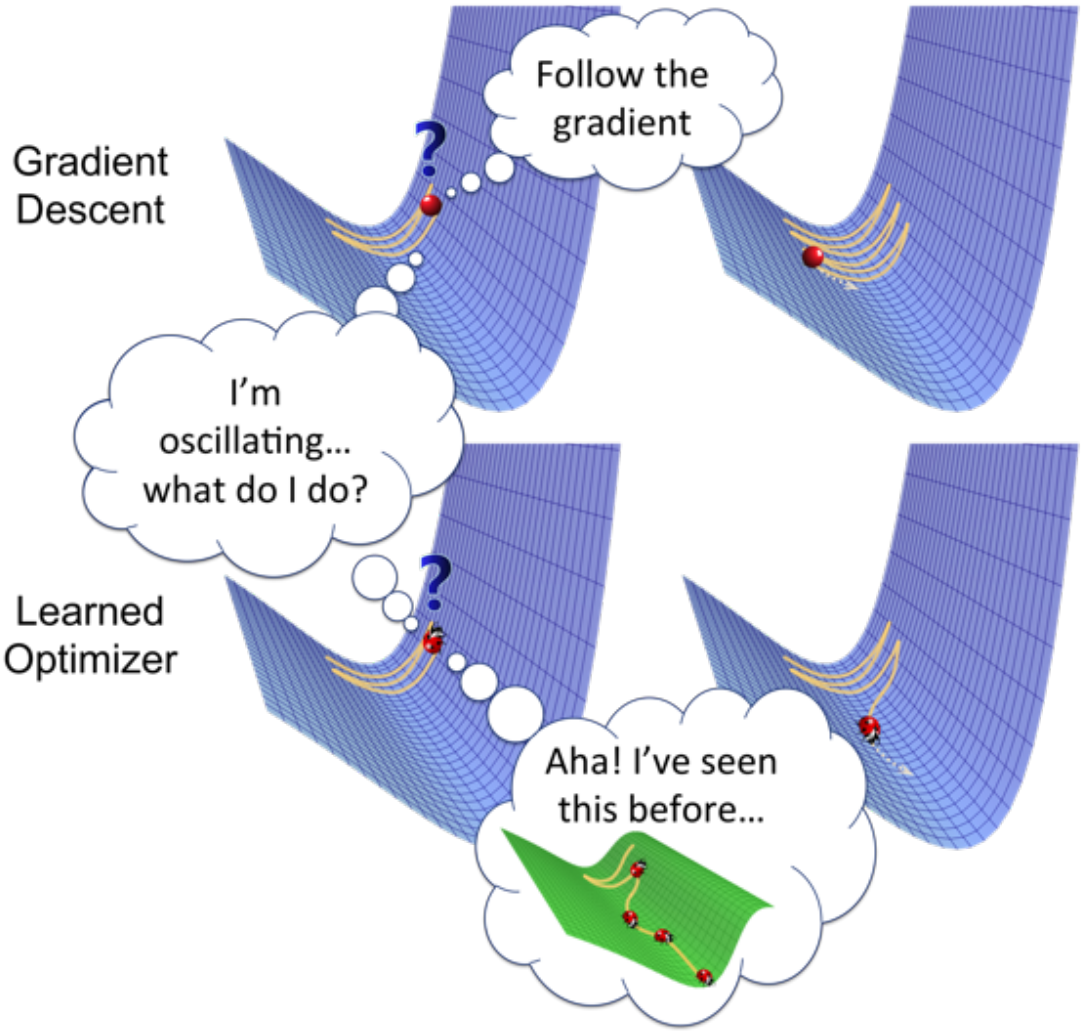
$\rightsquigarrow$ $\mathcal{L}_{\text{OI}}$ and $\mathcal{L}_{\text{EI}}$ perform best

$\rightsquigarrow$ $\mathcal{L}_{\text{OI}}$ easier to compute than $\mathcal{L}_{\text{EI}}$
because we need a predictive model to compute EI

Learning to Optimize by Li and Malik '17

Learning to Optimize via Reinforcement Learning

[Li and Malik'17]



Source: `https://bair.berkeley.edu/blog/2017/09/12/learning-to-optimize-with-rl/`

Learning to Optimize via Reinforcement Learning

[Li and Malik'17]

Reinforcement Learning for Learning to Optimize

State current location, objective values and gradients evaluated at the current and past locations

Action Step update Δx

Transition $x_t \leftarrow x_{t-1} + \Delta x$

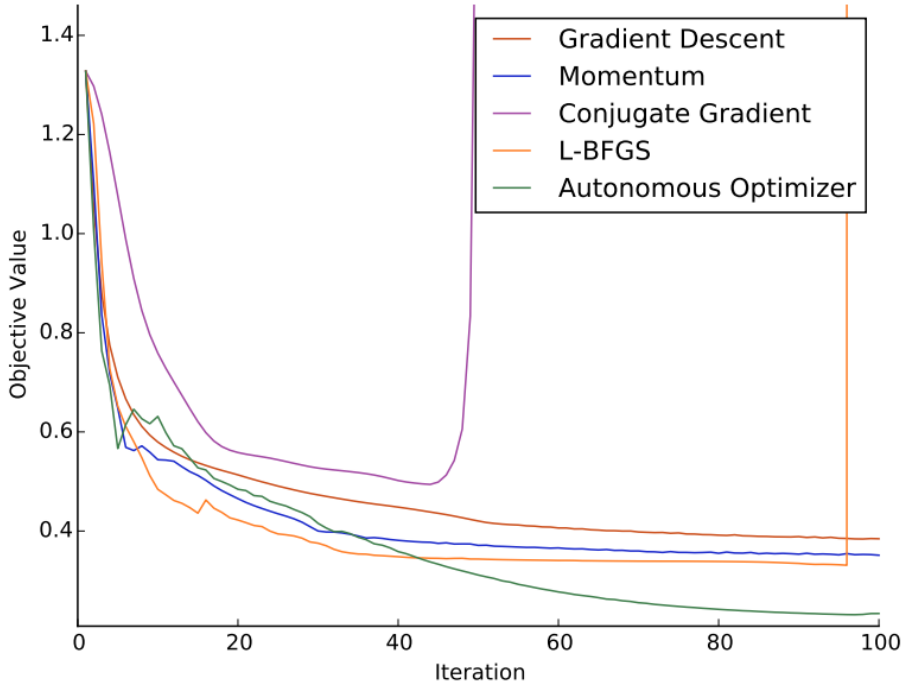
Cost/Reward Objective value at the current location

- Since the RL agent will optimize the cumulative cost, this is equivalent to $\mathcal{L}_{\text{sum}}$
- encourages the policy to reach the minimum of the objective function as quickly as possible

Policy DNN predicting μ_d of Gaussian (with constant variance σ^2) for dimension d; sample $\Delta x_d \sim \mathcal{N}(\mu_d, \sigma^2)$

Training Set randomly generated objective functions

Learning to Optimize via Reinforcement Learning
Results [Li and Malik'17]



- 2-layer DNN with ReLUs
- Training datasets for training RL agent:
  four multivariate Gaussians and sampling 25 points from each
  - $\rightsquigarrow$ hard toy problem

Meta-Learning Acquisition Functions for Bayesian Optimization by Volpp et al '19

Learning Acquisition Functions [Volpp et al.'19]

- Instead of learning everything, it might be sufficient to learn hand-design heuristics
- In Bayesian Optimization (BO), the most critical hand-design heuristic is the acquisition function
  - trade-off between exploitation and exploration
  - Depending on the problem at hand, you might need a different acquisition function
  - Choices:
    - probability of improvement (PI)
    - expected improvement (EI)
    - upper confidence bounds (UCB)
    - entropy search (ES) – quite expensive!
    - knowledge gradient (KG)
    - ...
- Idea: Learn a *neural acquisition function* from data

$\rightsquigarrow$ Replace acquisition function

Bayesian Optimization: Algorithm

Algorithm 1: Bayesian Optimization (BO)

Input : Search Space $\mathcal{X}$, black box function f, acquisition function α, maximal number of function evaluations m

$\mathcal{D}_0 \leftarrow$ initial_design($\mathcal{X}$);
for $n = 1, 2, \ldots m - |D_0|$ **do**
  $\hat{f} : \lambda \mapsto y \leftarrow$ fit predictive model on $\mathcal{D}_{n-1}$;
  select x_n by optimizing $x_n \in \arg\max_{x \in \mathcal{X}} \alpha(x; \mathcal{D}_{n-1}, \hat{f})$;
  Query $y_n := f(x_n)$;
  Add observation to data $D_n := D_{n-1} \cup \{\langle x_n, y_n \rangle\}$;
return *Best* x *according to* D_m *or* $\hat{f}$

Neural Acquisition Function [Volpp et al.'19]

Although the acquisition function α depends on the history $\mathcal{D}_{n-1}$ and the predictive model $\hat{f}$, α mainly makes use of the predictive mean μ and variance σ^2.

Neural acquisition function (AF):

$$\alpha_\theta(x) = \alpha_\theta(\mu_t(x), \sigma_t(x))$$

where θ are the parameters of a neural network,
and μ and $sigma$ are its inputs.

- Since the input is not x, it allows to learn scalable acquisition function
- No calibration of hyperparameter necessary, once the neural AF is learnt

RL to train Neural AF [Volpp et al.'19]

Policy π_θ: Neural acquisition function α_θ

Episode: run of π on $f \in \mathcal{F}'$

- $\mathcal{F}$ is a set of functions we can sample functions from

State s_t: μ_t and σ_t on a set of points ξ_t

Action a_t: Sampled point x_t

Reward r_t: negative simple regret: $r_t = f(x^*) - f(\hat{x})$

- assumes that we can estimate the optimal x^* for *training* functions

Transition probability : Noisy evaluation of f and the predictive model update

State [Volpp et al.'19]

- The state is described by a discrete set of points $\xi_t = \{\xi_n\}_{n=1}^N$
- We feed these points through the predictive model and the neural AF to obtain $\alpha_\theta(\xi_i) = \alpha_\theta(\mu_t(\xi_i), \sigma_t(\xi_i))$
- $\alpha_\theta(\xi_i)$ are interpreted as the logits of multinomial distribution, s.t.

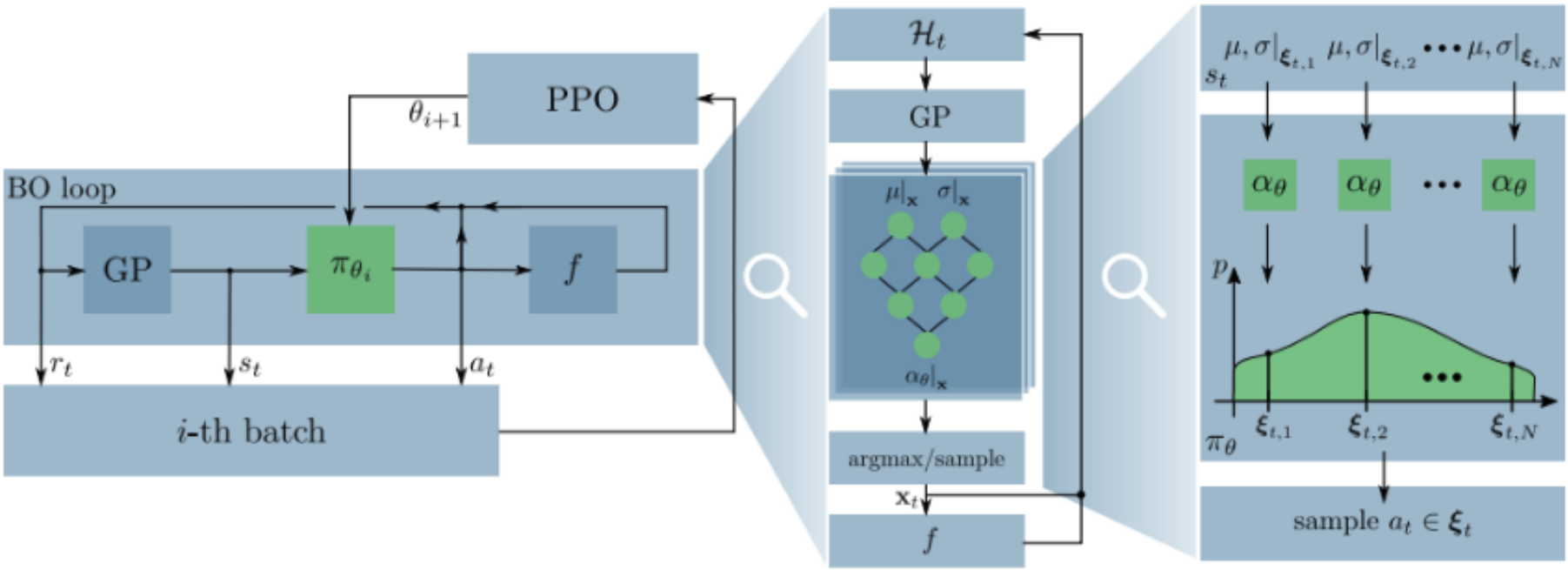
$$\pi_\alpha(\cdot \mid s_t) = \mathsf{Mult}\left[\alpha_\theta(\xi_1), \ldots, \alpha_\theta(\xi_N)\right]$$

- Due to curse of dimensionality, we need a two step approach for ξ_t
  1. sample ξ_{global} using a coarse Sobol grid
  2. sample ξ_{local} using local optimization starting from the best samples in ξ_{global}

$\rightsquigarrow$ $\xi_t = \xi_{\text{global}} \cup \xi_{\text{lokal}}$

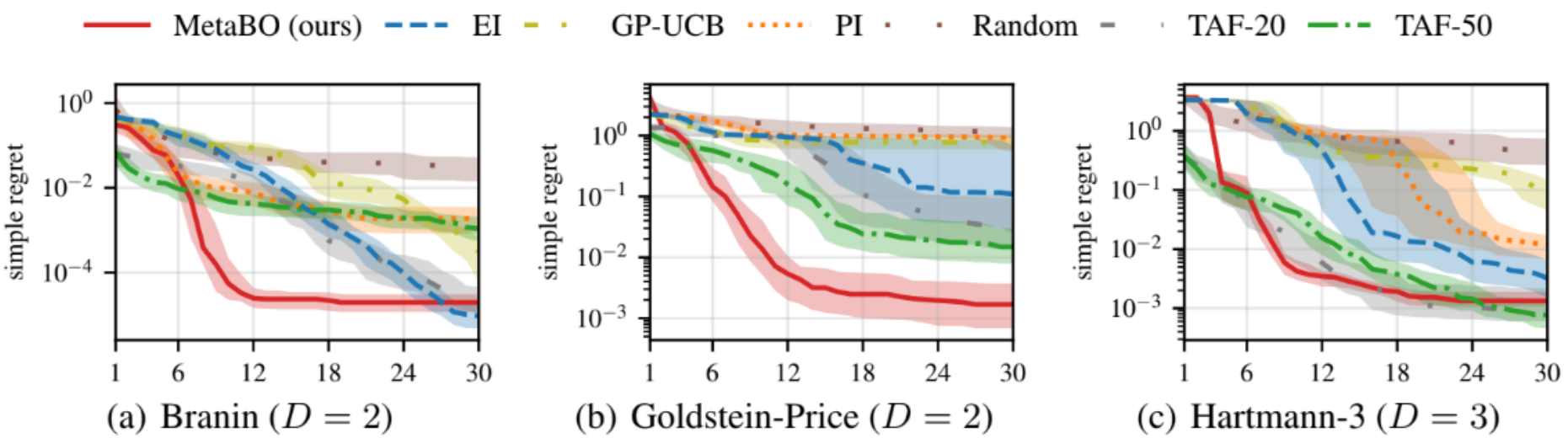
Learning Acquisition Functions

Results [Volpp et al.'19]



MetaBO Training Loop | Neural AF in the BO loop | Policy architecture

Results on Artificial Functions [Volpp et al.'19]



- Approach by [Volpp et al. '19] called MetaBO
- MetaBO performs better than other acquisition functions (EI, GP-UBC, PI) and other baselines (Random, TAF)

Assumption: You have a family of functions at hand that resembles your target functions.

Learning Goals

After these lectures, you are able to ...

- explain the high-level principles of meta-learning and its facets
- use algorithm selection to predict the best algorithm for a dataset
- improve the performance of AutoML tools by warmstarting its search
- transfer knowledge between DNNs by determining initial weights
- learn a DNN which can train another DNNs
- learn a policy to optimize a black-box function
- learn policies to either replace algorithm components or to control hyperparameters

Literature [These are links]

- [Automated Algorithm Selection: Survey and Perspectives]
- [MAML: Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks]
- [Efficient and Robust Automated Machine Learning]
- [Learning to learn by gradient descent by gradient descent]
- [Learning to Learn without Gradient Descent by Gradient Descent]
- [Learning to Optimize]
- [Meta-Learning Acquisition Functions for Bayesian Optimization]