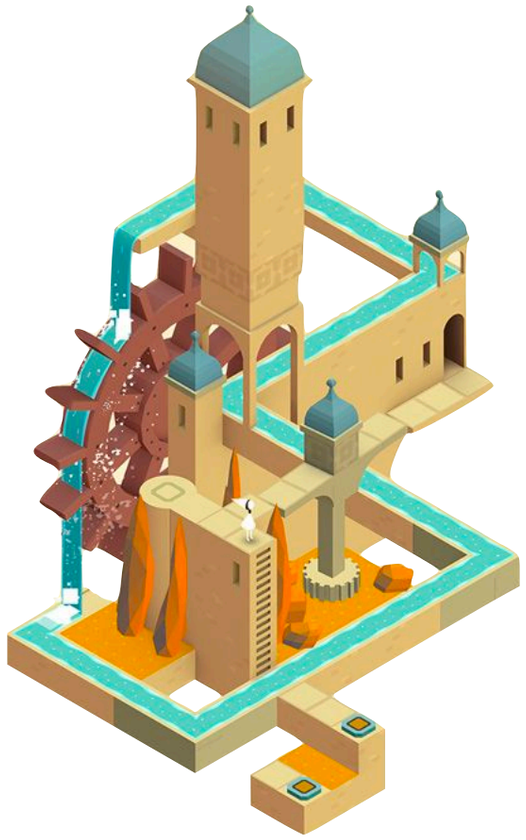
Advanced Course on Data Science & Machine Learning
Siena, 2019



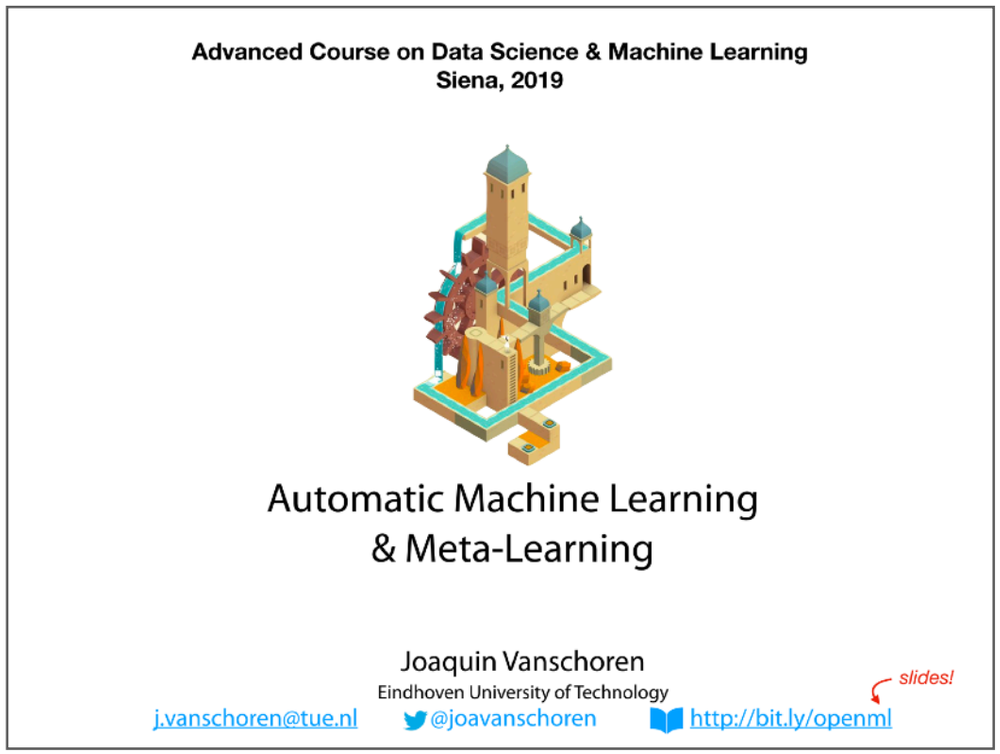
Automatic Machine Learning & Meta-Learning

part 1

Joaquin Vanschoren
Eindhoven University of Technology
j.vanschoren@tue.nl @joavanschoren http://bit.ly/openml

slides!

Slides / Book



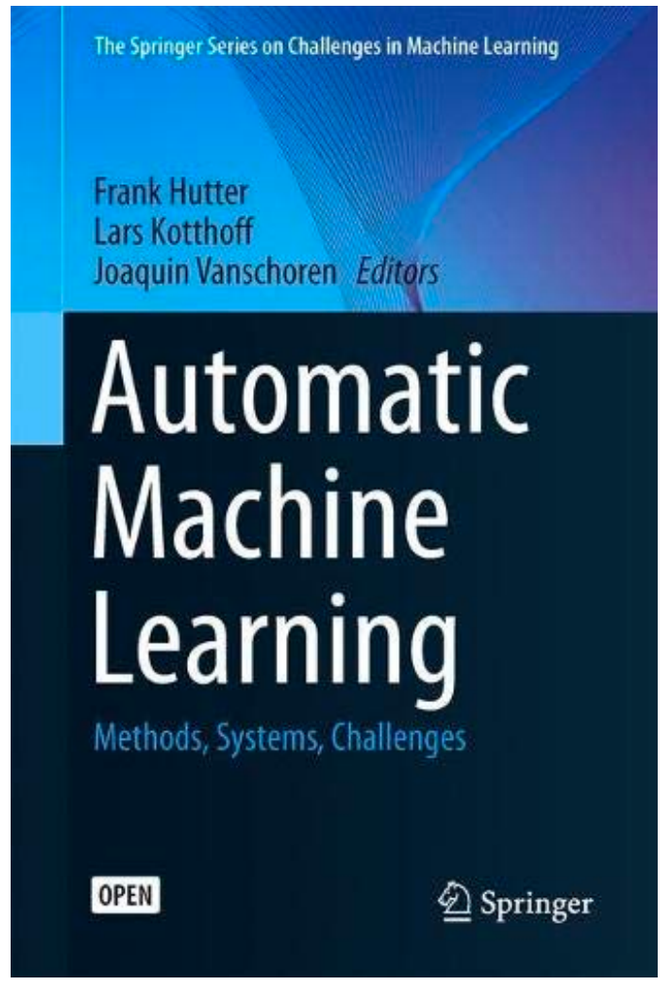
http://bit.ly/openml

More slides:

www.automl.org/events ->
AutoML Tutorial NeurIPS 2018

Video:

www.youtube.com/watch?v=0eBR8a4MQ30

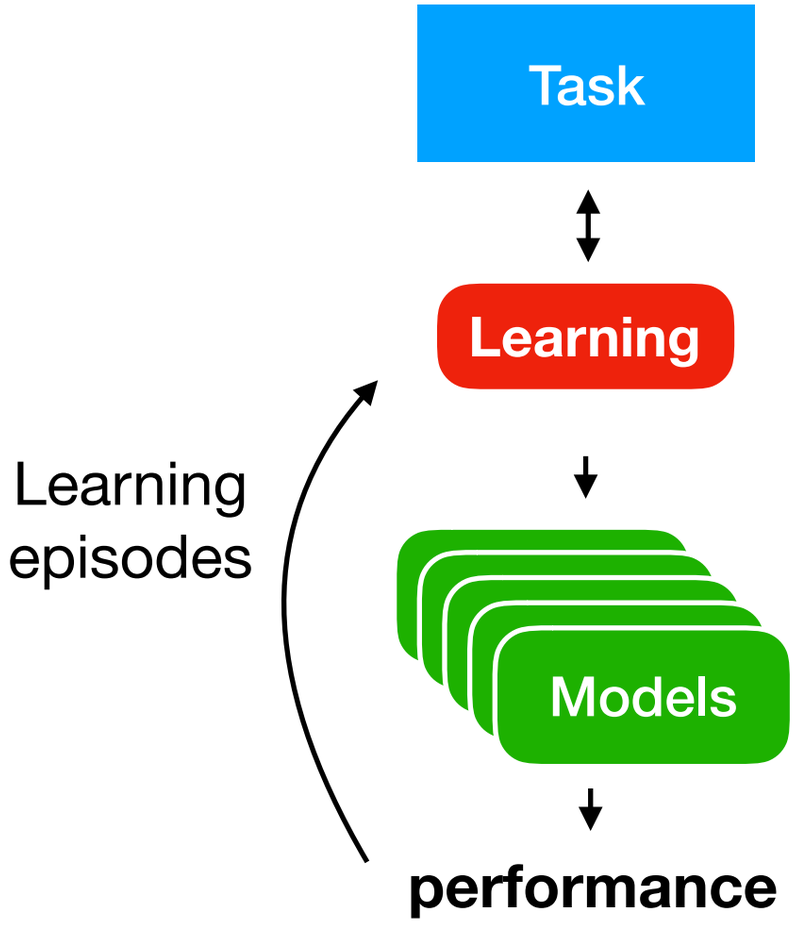


Open access book

PDF (free): www.automl.org/book
www.amazon.de/dp/3030053172

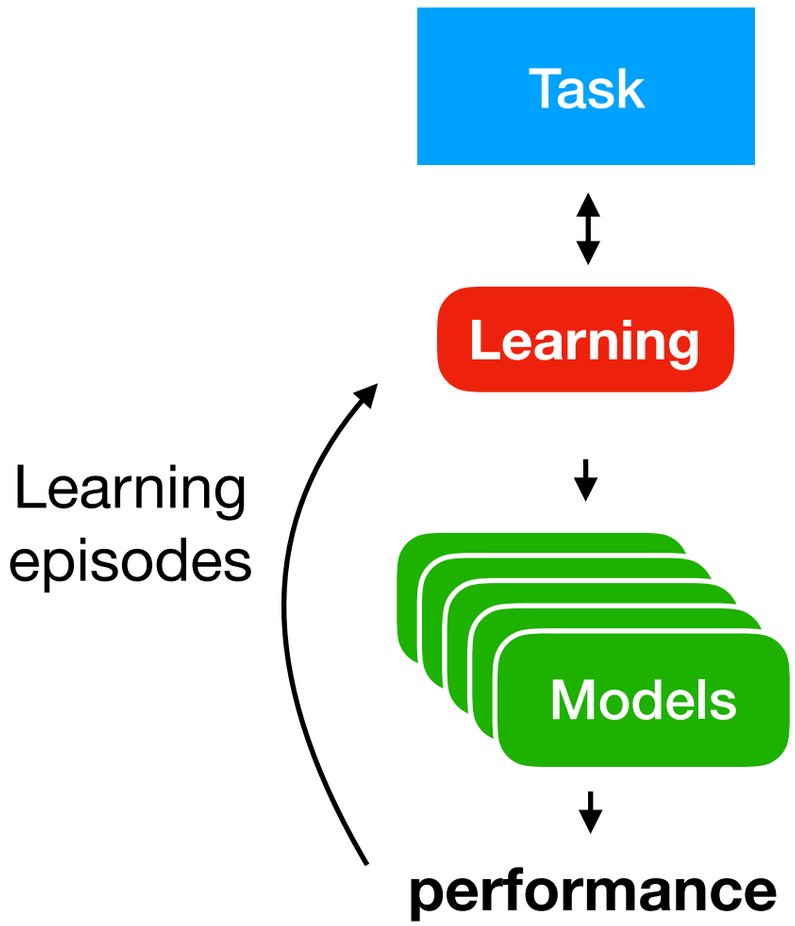
Learning takes experimentation

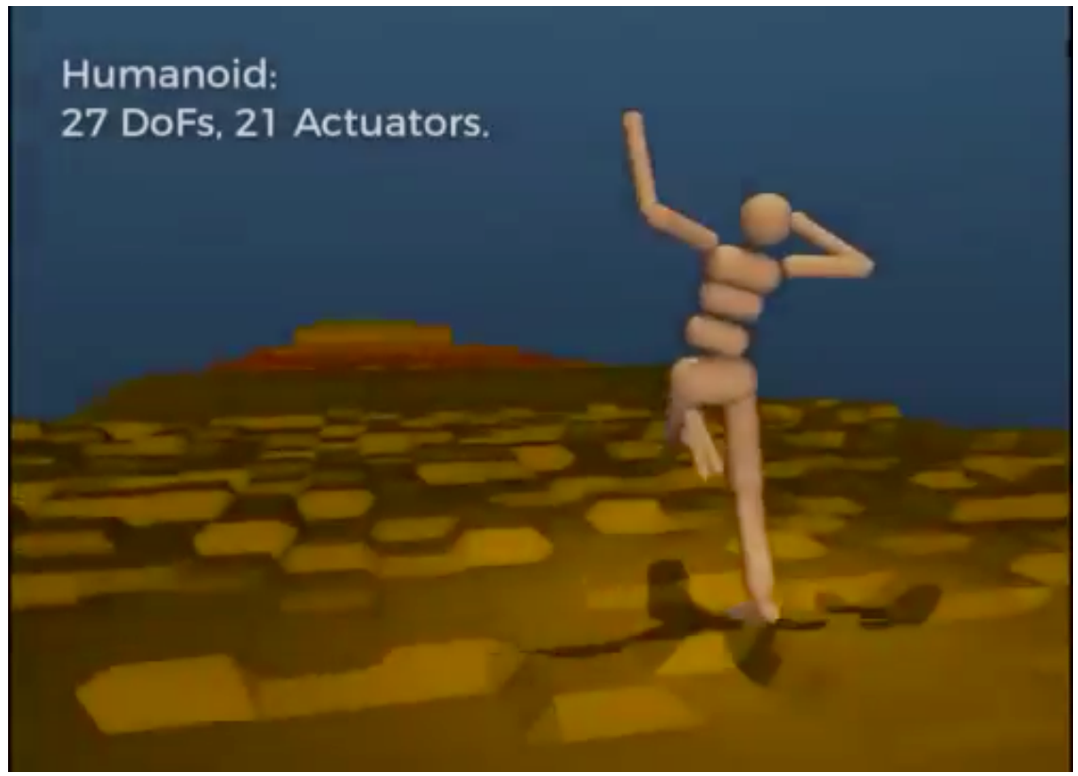
It can take a *lot* of trial and error



Learning takes experimentation

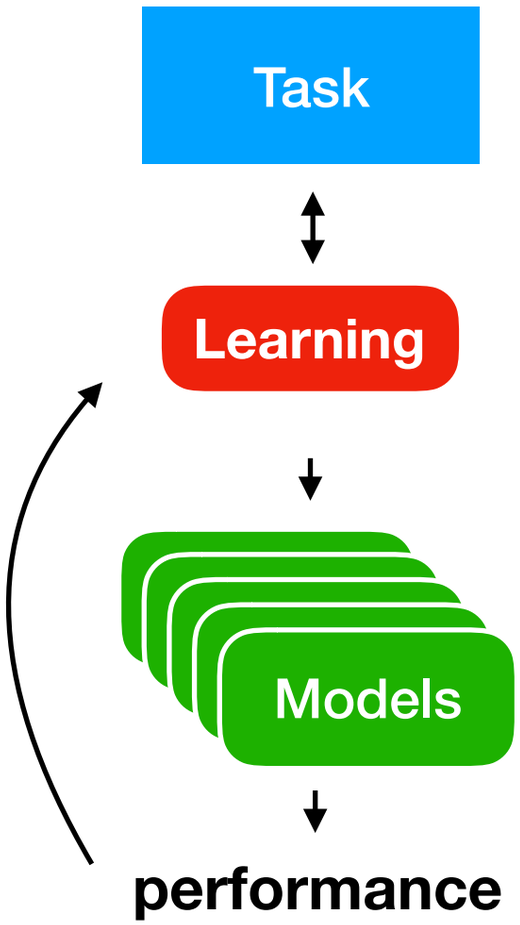
It can take a *lot* of trial and error



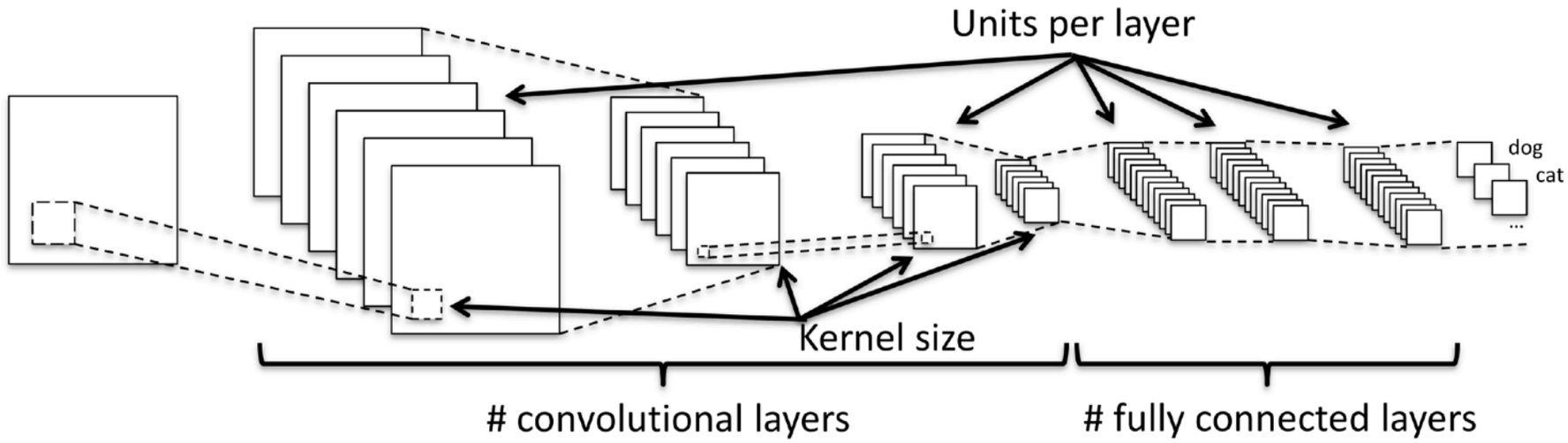


Learning to learn takes experimentation

Building machine learning systems requires a *lot* of expertise and trials



Neural networks



- design architecture (operators/layers)
- many sensitive hyperparameters
  - optimization algorithm, learning rate,…
  - regularization, dropout,…
  - batch size, epochs, early stopping, …
  - data augmentation, …

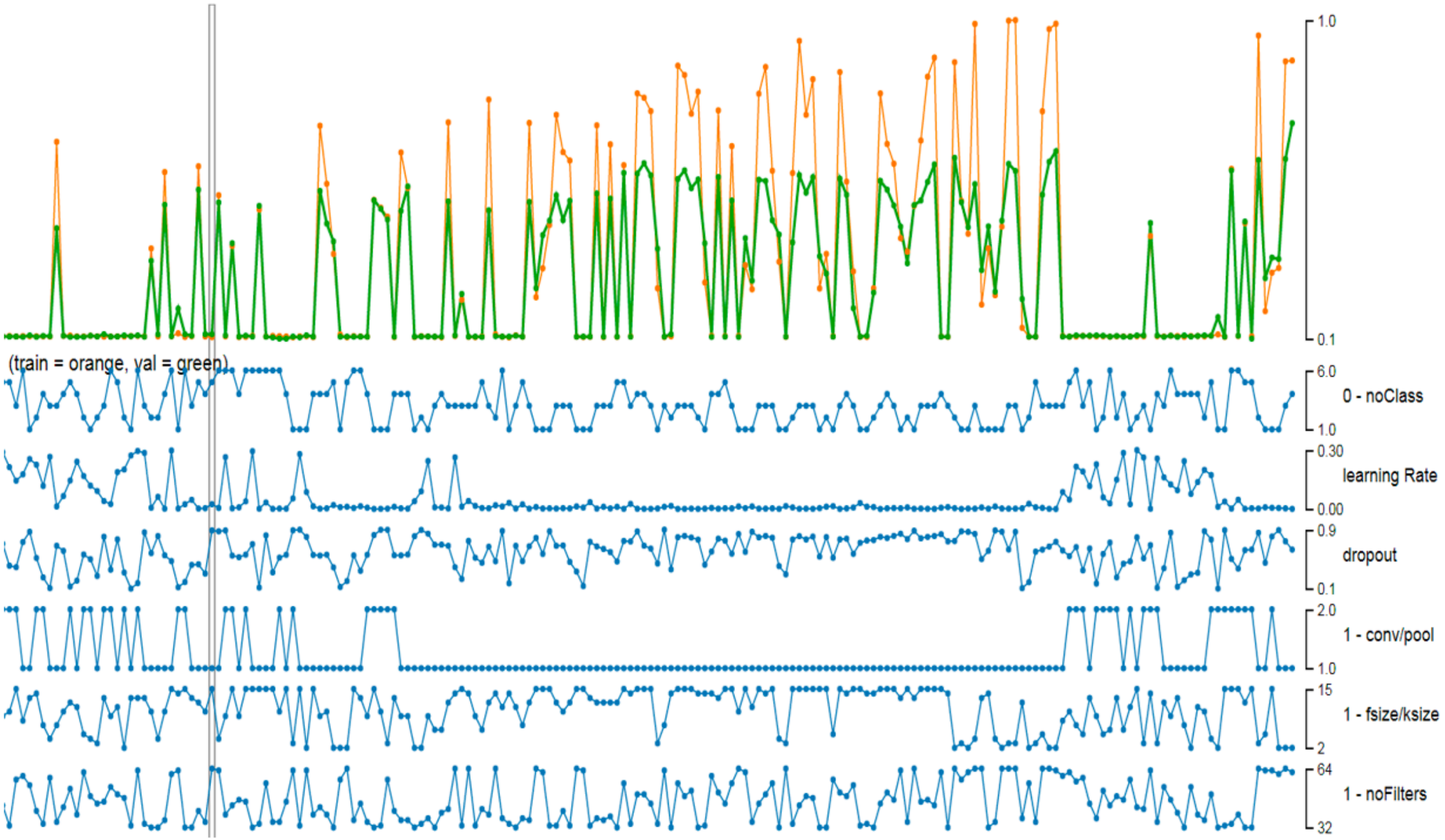
Hyperparameters

Every design decision made by the *user (architecture, operators, tuning,…)*

	Name	Range	Default	log scale	Type	Conditional
Network hyperparameters	batch size	$[32, 4096]$	32	✓	float	-
	number of updates	$[50, 2500]$	200	✓	int	-
	number of layers	$[1, 6]$	1	-	int	-
	learning rate	$[10^{-6}, 1.0]$	10^{-2}	✓	float	-
	L_2 regularization	$[10^{-7}, 10^{-2}]$	10^{-4}	✓	float	-
	dropout output layer	$[0.0, 0.99]$	0.5	✓	float	-
	solver type	{SGD, Momentum, Adam, Adadelta, Adagrad, smorm, Nesterov }	smorm3s	-	cat	-
	lr-policy	{Fixed, Inv, Exp, Step}	fixed	-	cat	-
Conditioned on solver type	β_1	$[10^{-4}, 10^{-1}]$	10^{-1}	✓	float	✓
	β_2	$[10^{-4}, 10^{-1}]$	10^{-1}	✓	float	✓
	ρ	$[0.05, 0.99]$	0.95	✓	float	✓
	momentum	$[0.3, 0.999]$	0.9	✓	float	✓
Conditioned on lr-policy	γ	$[10^{-3}, 10^{-1}]$	10^{-2}	✓	float	✓
	k	$[0.0, 1.0]$	0.5	-	float	✓
	s	$[2, 20]$	2	-	int	✓
Per-layer hyperparameters	activation-type	{Sigmoid, TanH, ScaledTanH, ELU, ReLU, Leaky, Linear}	ReLU	-	cat	✓
	number of units	$[64, 4096]$	128	✓	int	✓
	dropout in layer	$[0.0, 0.99]$	0.5	-	float	✓
	weight initialization	{Constant, Normal, Uniform, Glorot-Uniform, Glorot-Normal, He-Normal, He-Uniform, Orthogonal, Sparse}	He-Normal	-	cat	✓
	std. normal init.	$[10^{-7}, 0.1]$	0.0005	-	float	✓
	leakiness	$[0.01, 0.99]$	$\frac{1}{3}$	-	float	✓
	tanh scale in	$[0.5, 1.0]$	$2/3$	-	float	✓
	tanh scale out	$[1.1, 3.0]$	1.7159	✓	float	✓

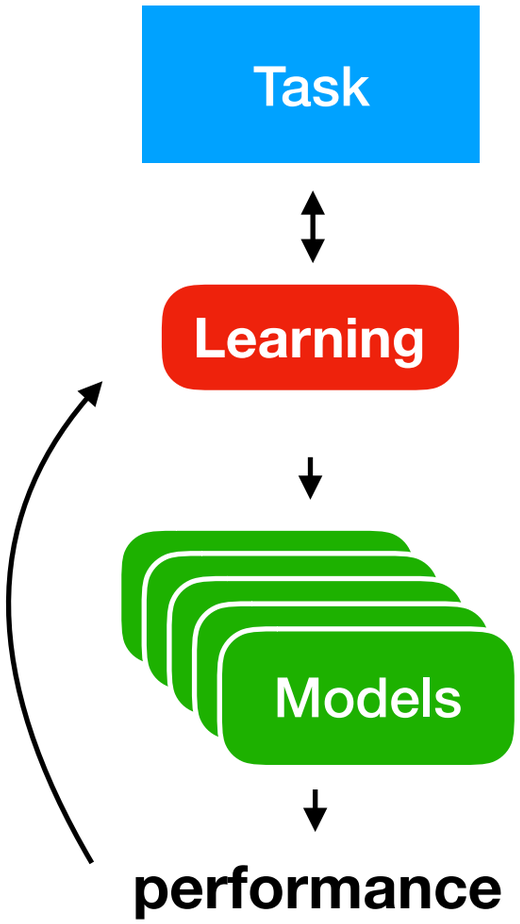
Hyperparameters

Can be very sensitive

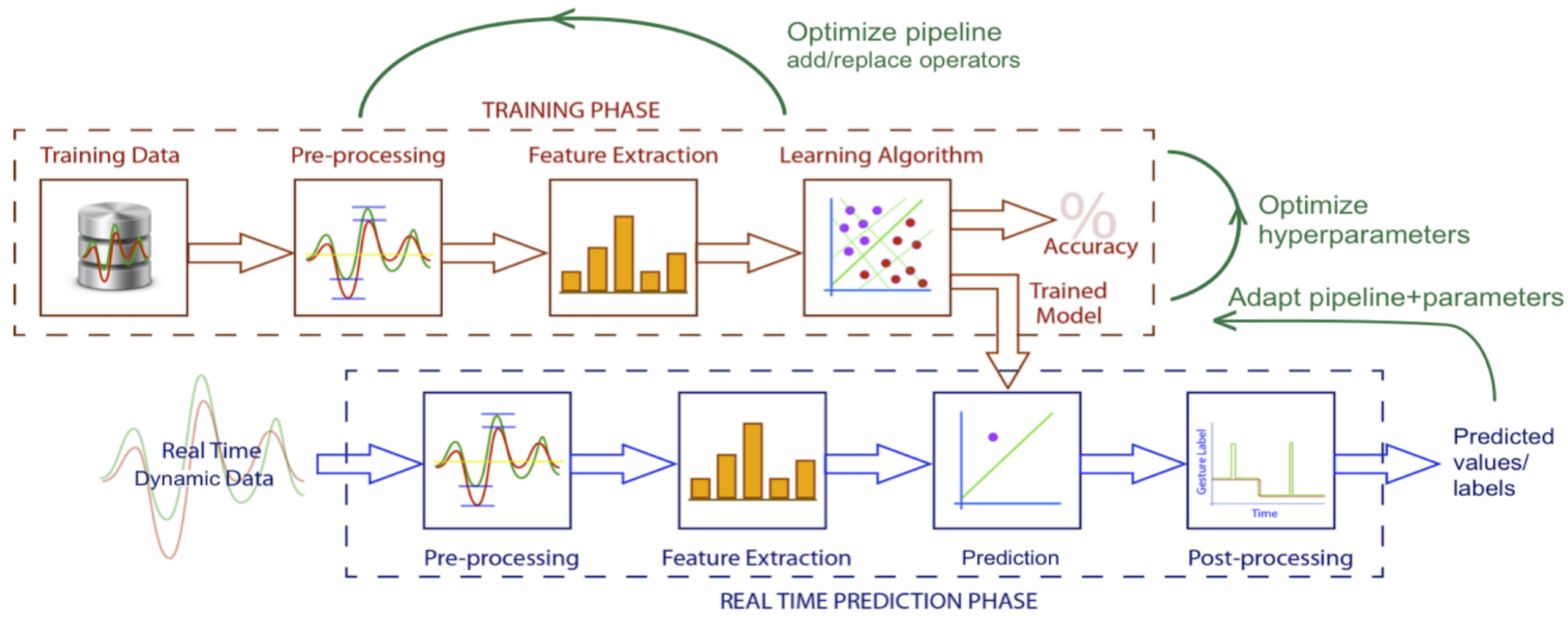


Learning to learn takes experimentation

Building machine learning systems requires a lot of expertise and trials



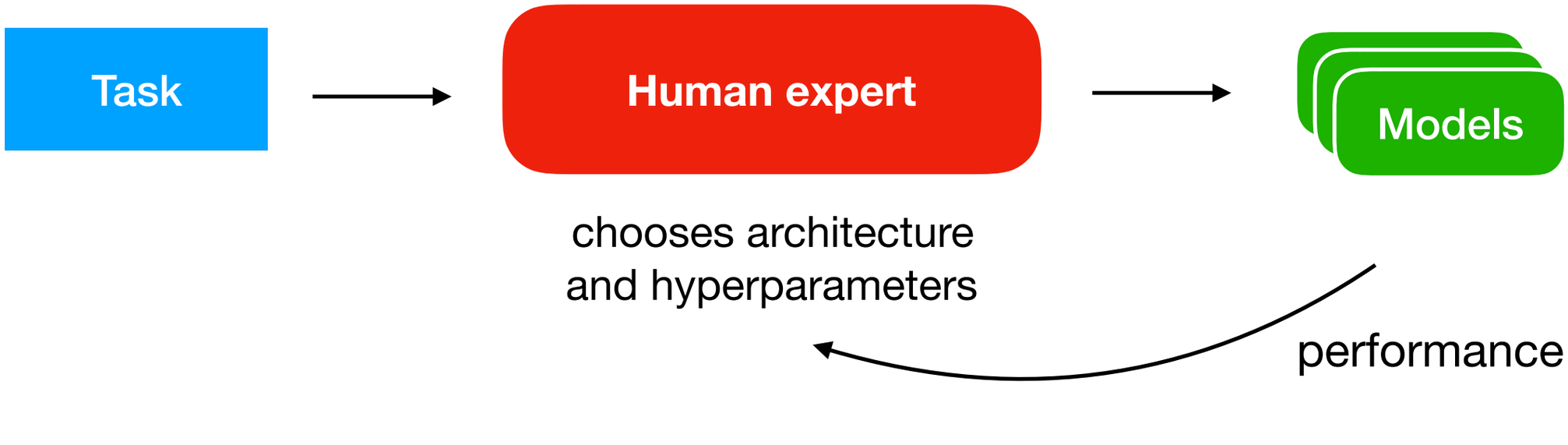
Machine learning pipelines



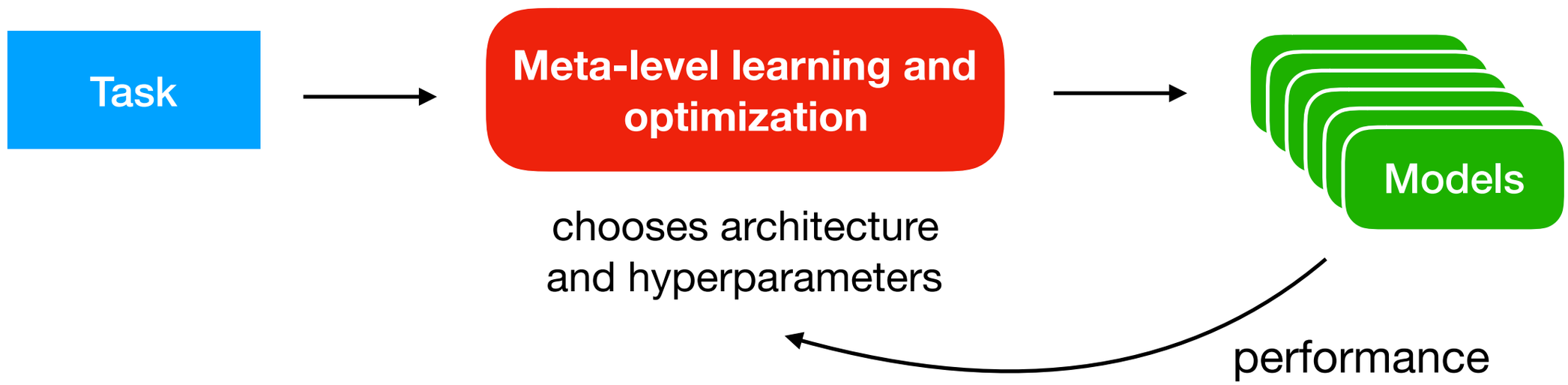
- design pipeline architecture (operators)
- clean, preprocess data (!)
- select and/or engineer features
- select and tune models
- adjust to evolving input data (concept drift),…

Automated machine learning

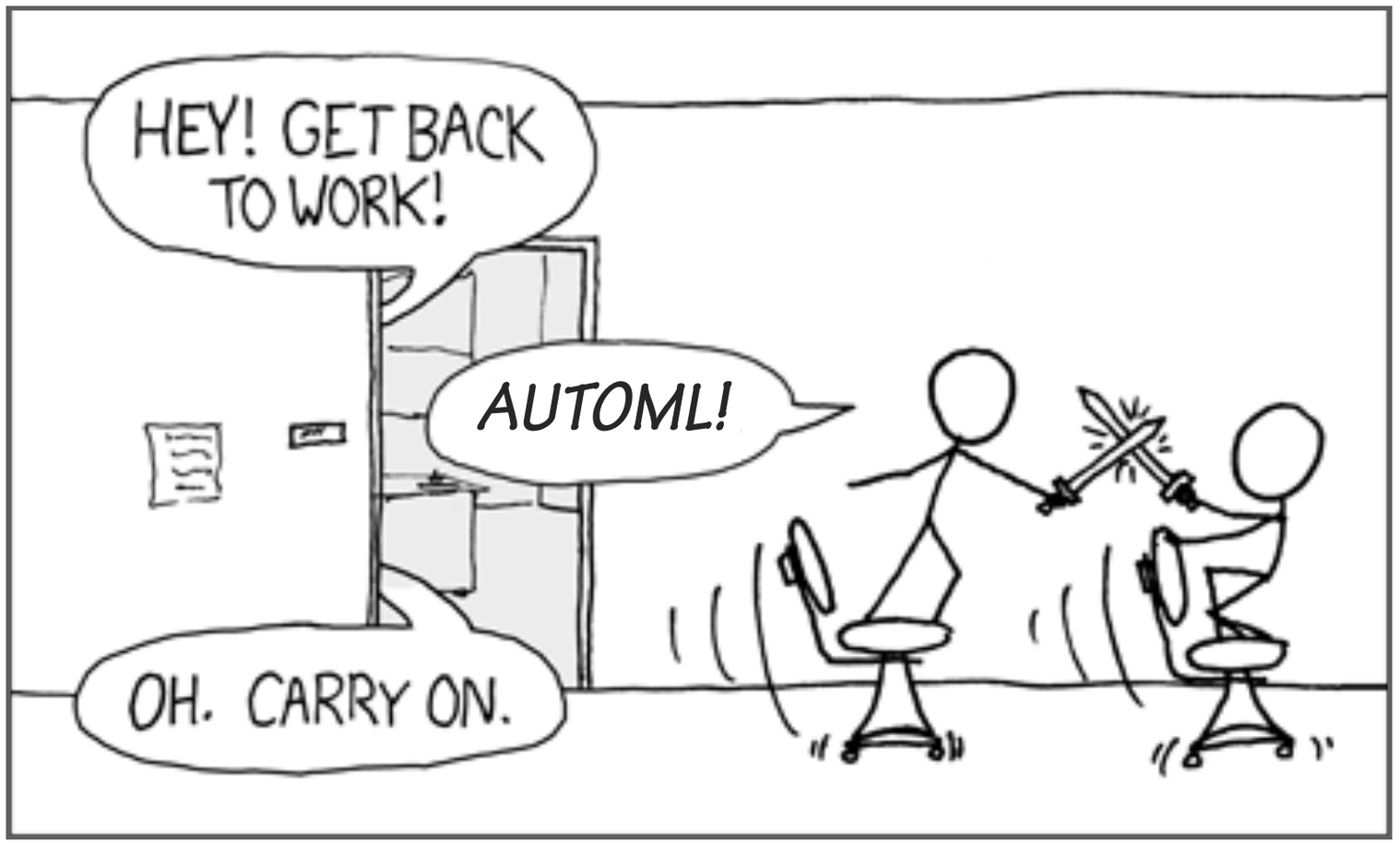
Current practice



Replace manual model building by automation



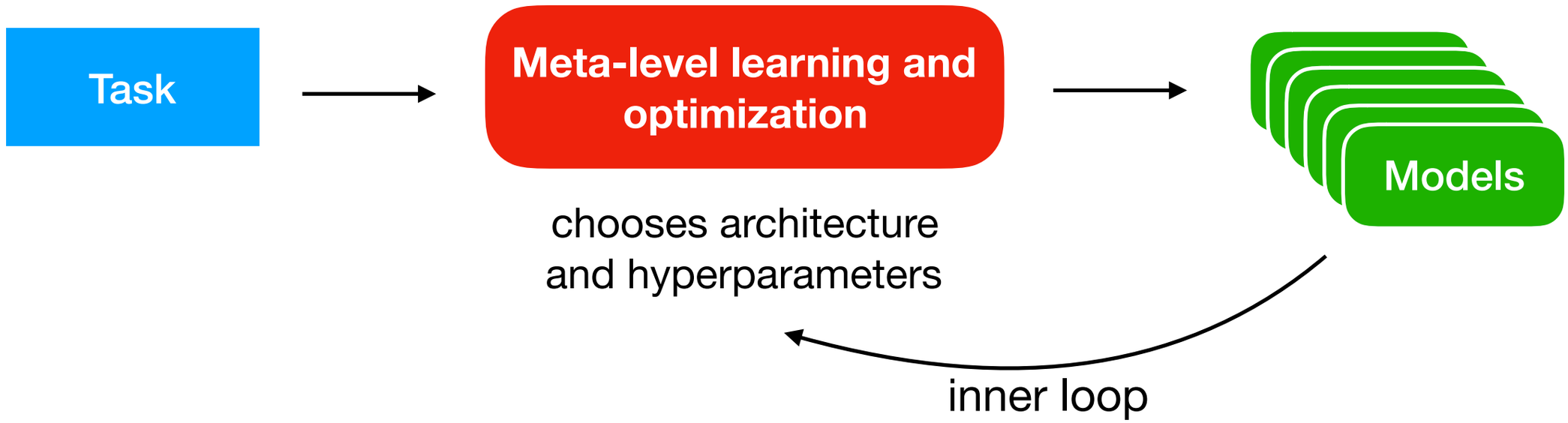
THE DATA SCIENTIST'S #1 EXCUSE FOR LEGITIMATELY SLACKING OFF: "THE AUTOML TOOL IS OPTIMIZING MY MODELS!"



Sutton 2019

Human-in-the-loop AutoML (semi-AutoML)

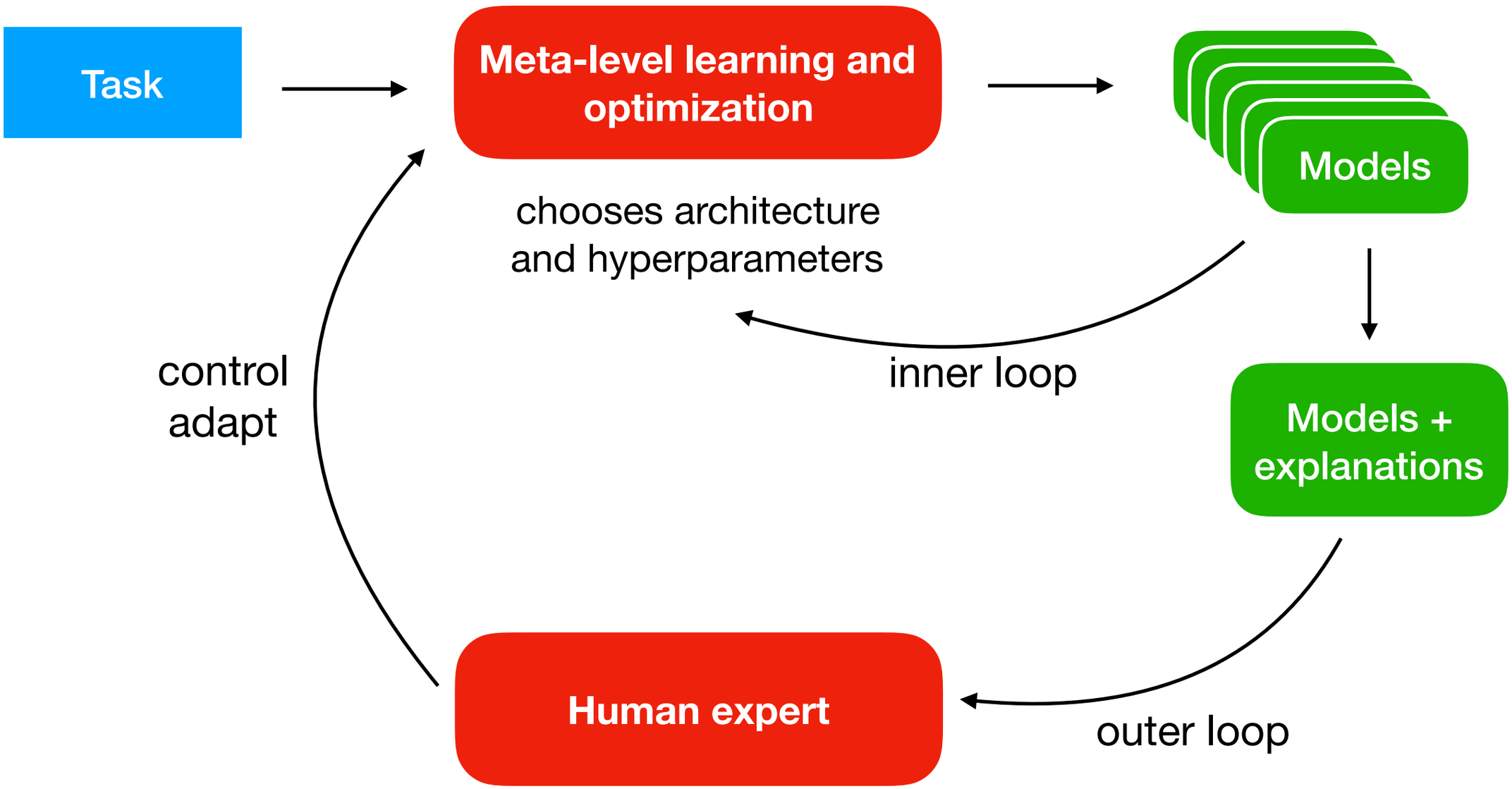
Domain knowledge and human expertise are very valuable
e.g. unknown unknowns, preference learning,…



Sutton 2019

Human-in-the-loop AutoML (semi-AutoML)

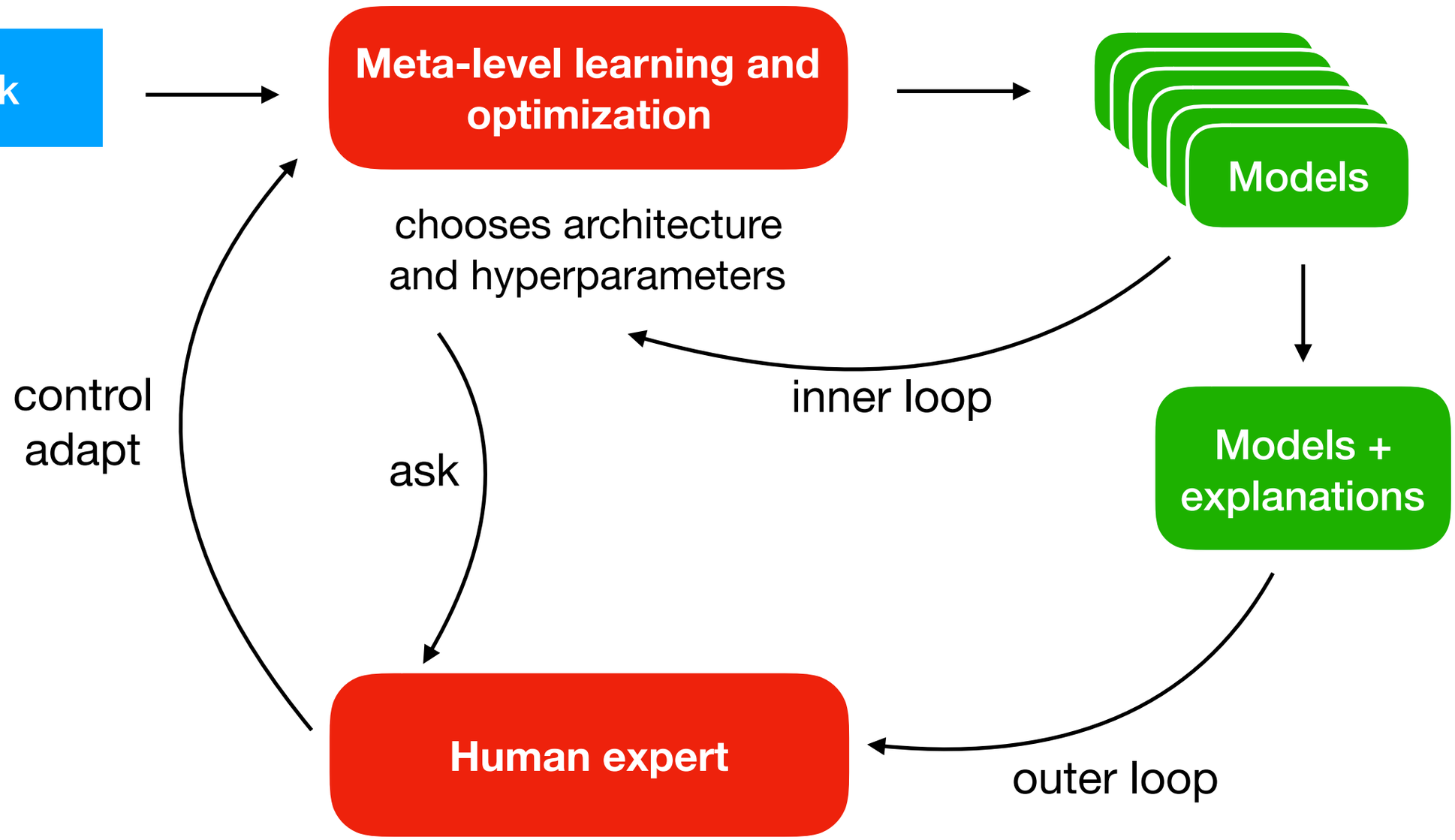
Domain knowledge and human expertise are very valuable
e.g. unknown unknowns, preference learning,…



Sutton 2019

Human-in-the-loop AutoML (semi-AutoML)

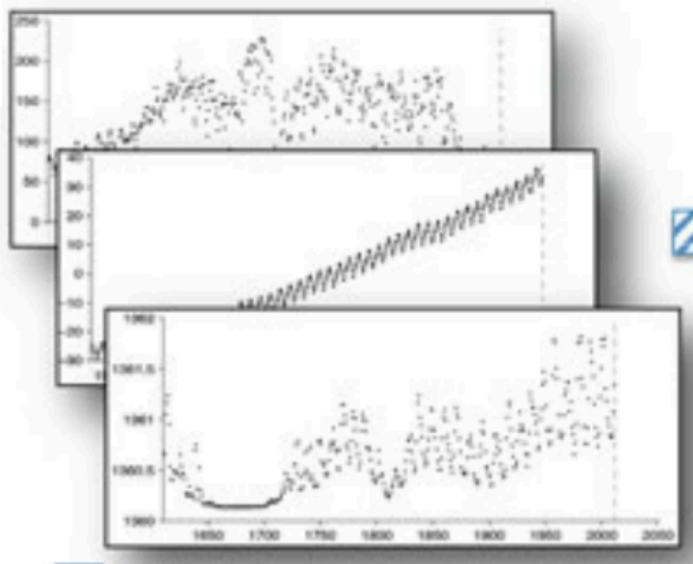
Domain knowledge and human expertise are very valuable
e.g. unknown unknowns, preference learning,…



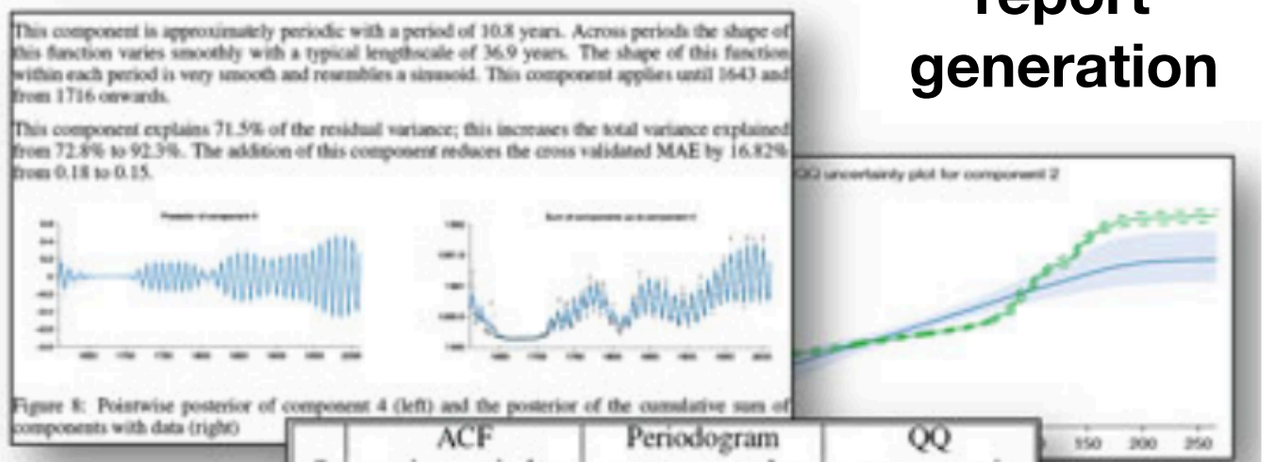
Human-in-the-loop AutoML (semi-AutoML)

Explain model in human language: *automatic statistician*

data input



report generation

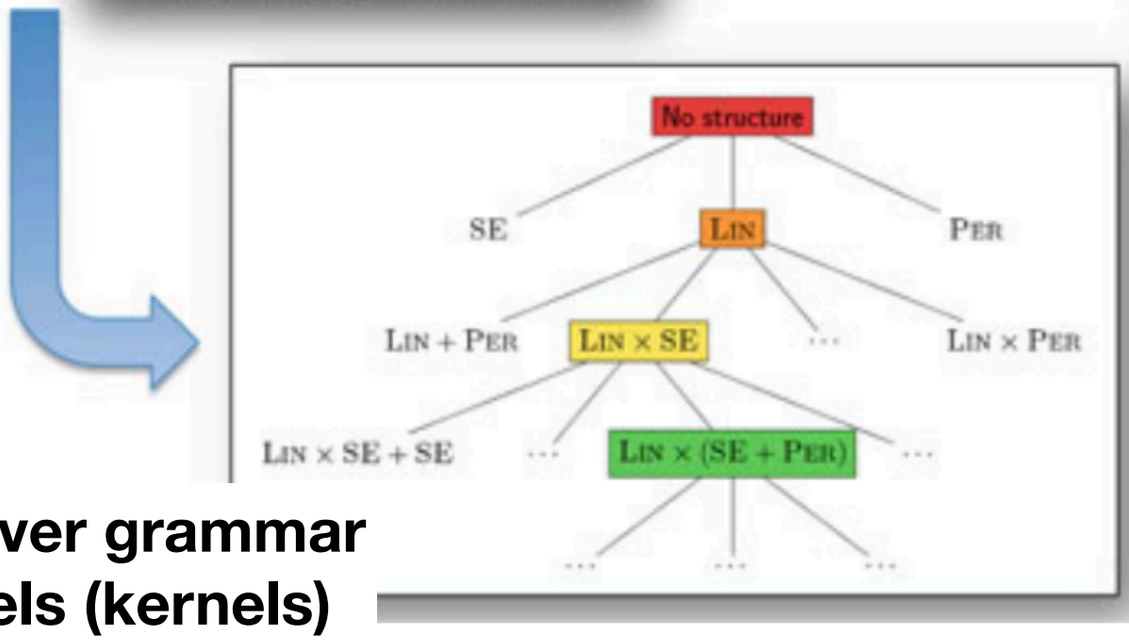


This component is approximately periodic with a period of 10.8 years. Across periods the shape of this function varies smoothly with a typical lengthscale of 36.9 years. The shape of this function within each period is very smooth and resembles a sinusoid. This component applies until 1643 and from 1716 onwards.

This component explains 71.5% of the residual variance; this increases the total variance explained from 72.8% to 92.3%. The addition of this component reduces the cross validated MAE by 16.82% from 0.18 to 0.15.

Figure 8: Pointwise posterior of component 4 (left) and the posterior of the cumulative sum of components with data (right)

#	ACF min	ACF min loc	Periodogram max	Periodogram max loc	QQ max	QQ min
1	0.502	0.582	0.341	0.413	0.341	0.679
2	0.802	0.199	0.558	0.630	**0.049**	0.785
3	0.251	0.475	0.799	0.447	0.534	0.769
4	0.527	0.503	0.504	0.481	0.430	0.616
5	0.493	0.477	0.503	0.487	0.518	0.381



search over grammar of models (kernels)

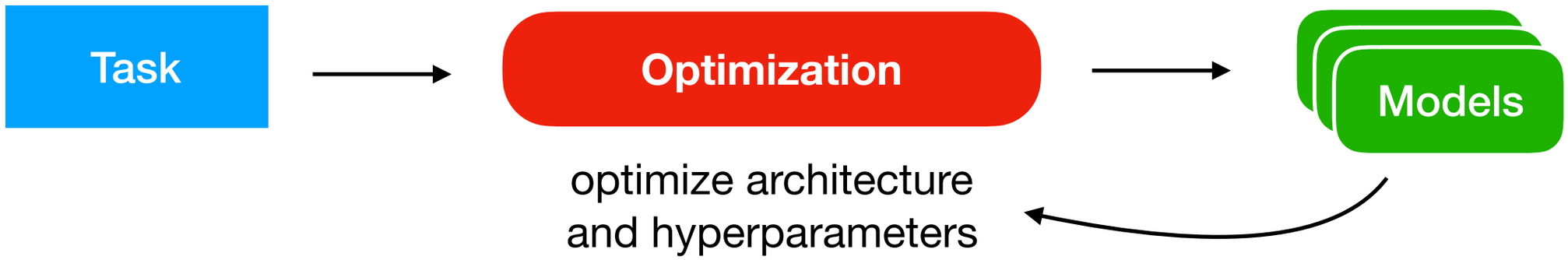
Lin
with linearly growing amplitude

SE × Per
approximately periodic function

model component to English translation

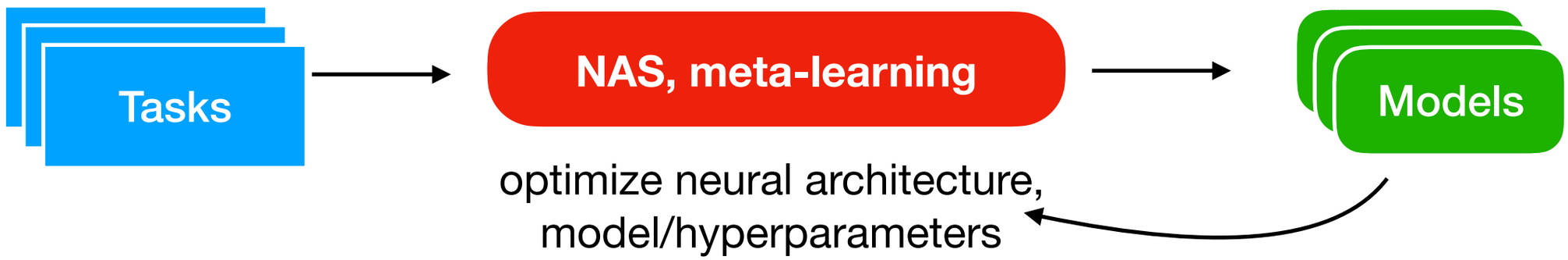
Overview

part 1 AutoML introduction, promising optimization techniques



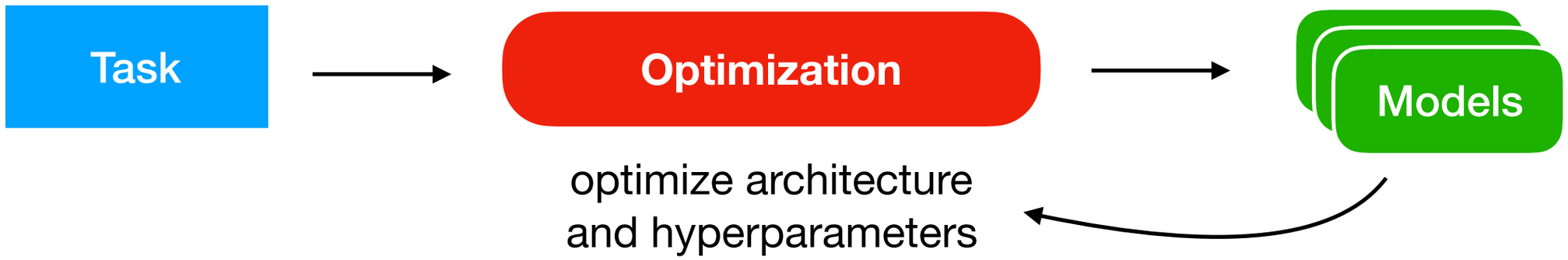
part 2 Meta-learning

part 3 Neural architecture search, meta-learning on neural nets



Overview

part I AutoML introduction, optimization techniques



1. Problem definition
2. (Pipeline) architecture search
3. Optimization techniques
4. Performance improvements
5. Benchmarks

AutoML Definition

Let:

$\{A^{(1)}, A^{(2)}, \ldots, A^{(n)}\}$ be a set of *algorithms* (operators)

$\Lambda^{(i)} = \lambda_1 \times \lambda_2 \times \ldots \times \lambda_m$ be the *hyperparameter space* for $A^{(i)}$

$\mathscr{A}$ be the space of possible *architectures* of one or more algorithms

$\Lambda_{\mathscr{A}} = \Lambda^{(1)} \times \Lambda^{(2)} \times \ldots \times \Lambda^{(m)}$ its combined *configuration space*

$\boldsymbol{\lambda} \in \Lambda_{\mathscr{A}}$ a specific *configuration* (of architecture and hyperparameters)

$\mathscr{L}(\boldsymbol{\lambda}, D_{train}, D_{valid})$ the loss of the model created by $\boldsymbol{\lambda}$, trained on data D_{train}, and validated on data D_{valid}

Find the configuration that minimizes the expected loss on a dataset $\mathscr{D}$:

$$\boldsymbol{\lambda}^* = \underset{\lambda \in \Lambda_{\mathscr{A}}}{\operatorname{argmin}} \mathbb{E}_{(D_{train}, D_{valid}) \sim \mathscr{D}} \mathscr{L}(\boldsymbol{\lambda}, D_{train}, D_{test})$$

Types of hyperparameters

- Continuous (e.g. learning rate, SVM_C,…)
- Integer (e.g. number of hidden units, number of boosting iterations,…)
- Categorical
  - e.g. choice of algorithm (SVM, RandomForest, Neural Net,…)
  - e.g. choose of operator (Convolution, MaxPooling, DropOut,…)
  - e.g. activation function (ReLU, Leaky ReLU, tanh,…)
- Conditional
  - e.g. SVM kernel if SVM is selected, kernel width if RBF kernel is selected
  - e.g. Convolution kernel size if Convolution layer is selected

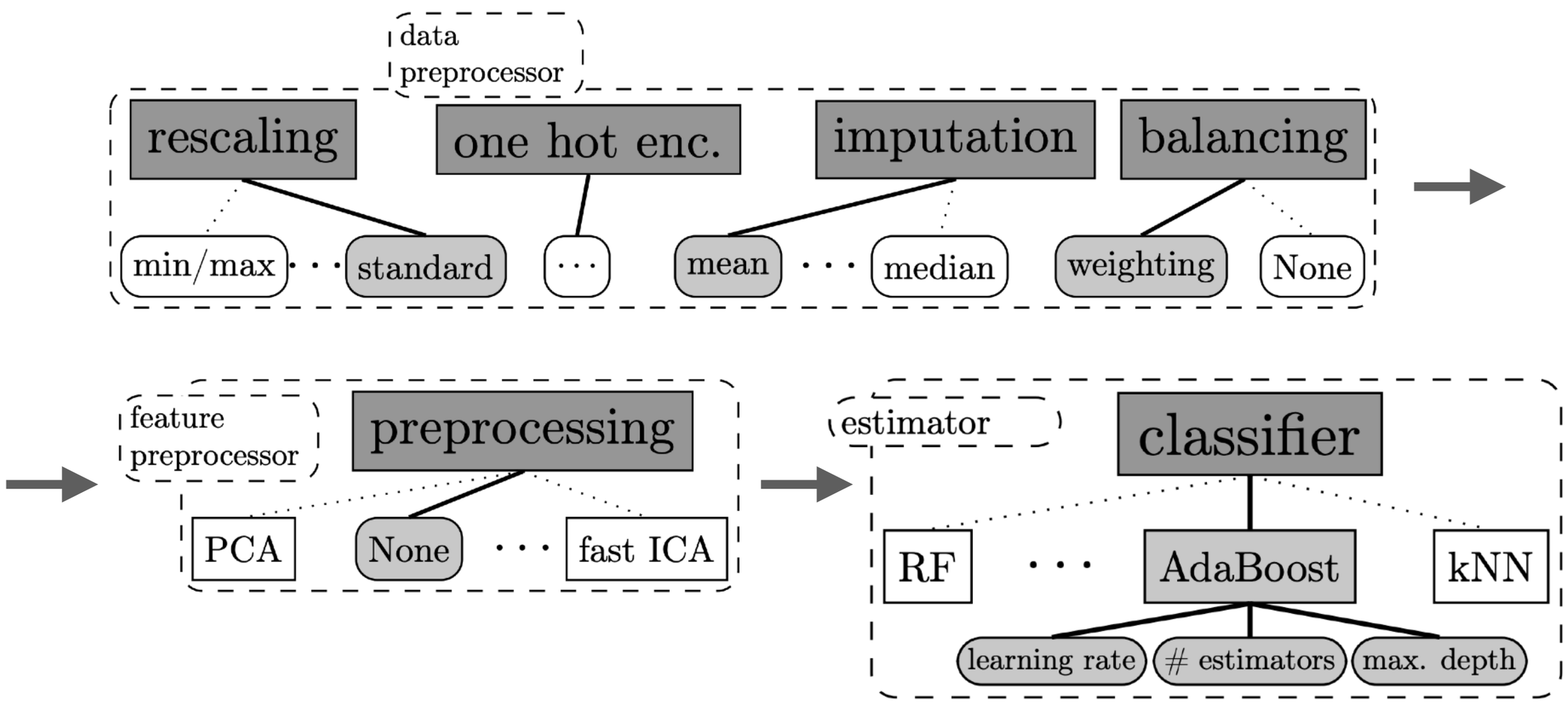
Architecture vs hyperparameters

- We can identify two subproblems:
  - **Architecture search**: search space of all possible architectures
    - ***Pipelines***: Fixed predefined pipeline, grammars, genetic programming, planning, Monte-Carlo Tree Search
    - ***Neural architectures***: See part 3
  - **Hyperparameter optimization**: optimize remaining hyperparameters
    - ***Optimization***: grid/random search, Bayesian optimization, evolution, multi-armed bandits, gradient descent (only NNs)
    - ***Meta-learning:*** See part 2
- Can be solved *consecutively*, *simultaneously* or *interleaved*
- ***Compositionality***: breaking down the learning process into smaller reusable tasks makes it easier, more transferable, more robust

Pipeline search: fixed pipelines

- Parameterize best-practice (linear) pipelines
  - Introduce conditional hyperparameters $\lambda_r \in \{A^{(1)}, \ldots, A^{(k)}, \varnothing\}$
  - Combined Algorithm Selection and Hyperparameter optimization (CASH):

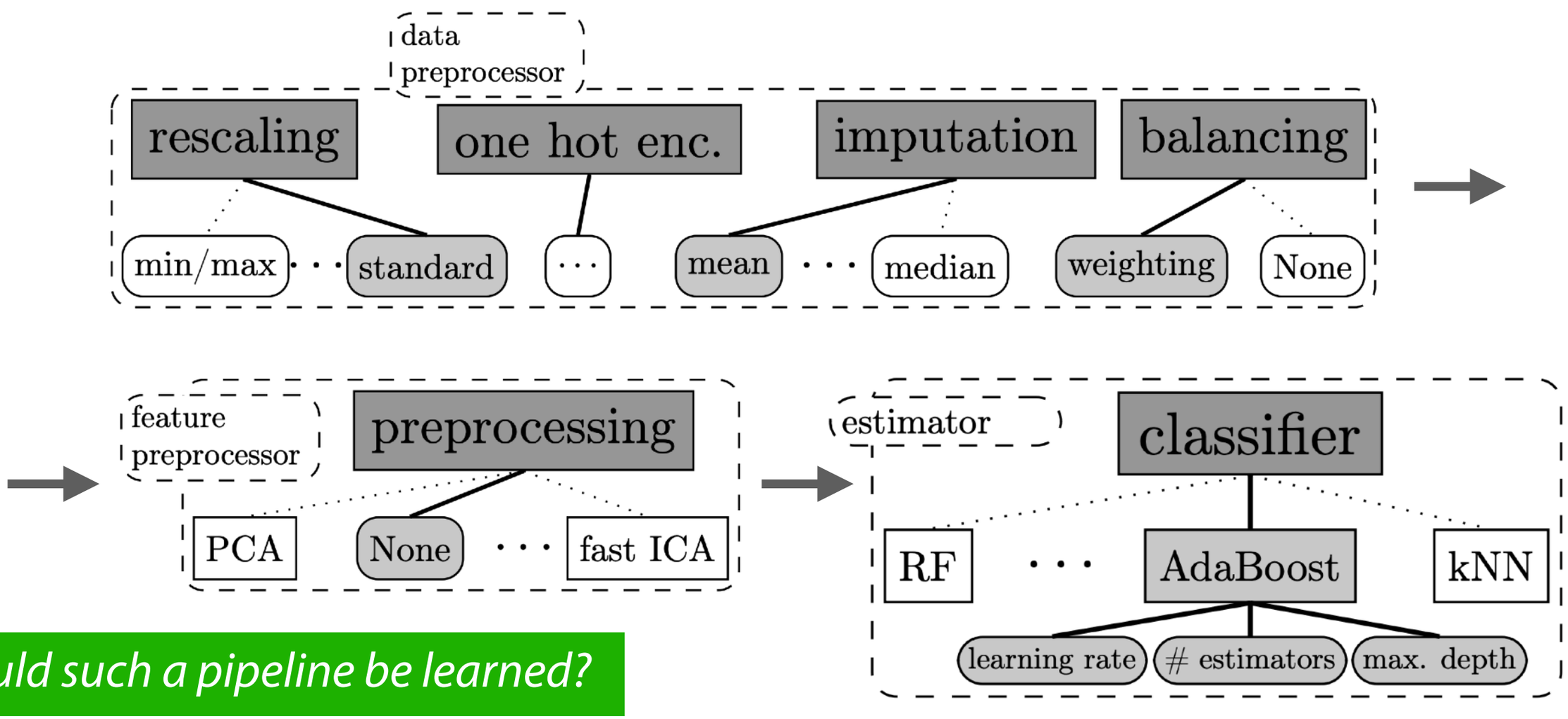
$$\Lambda_{\mathscr{A}} = \Lambda^{(1)} \times \Lambda^{(2)} \times \ldots \times \Lambda^{(m)} \times \lambda_r$$



Pipeline search: fixed pipelines

- Parameterize best-practice (linear) pipelines
  - Introduce conditional hyperparameters $\lambda_r \in \{A^{(1)}, \ldots, A^{(k)}, \varnothing\}$
  - Combined Algorithm Selection and Hyperparameter optimization (CASH):

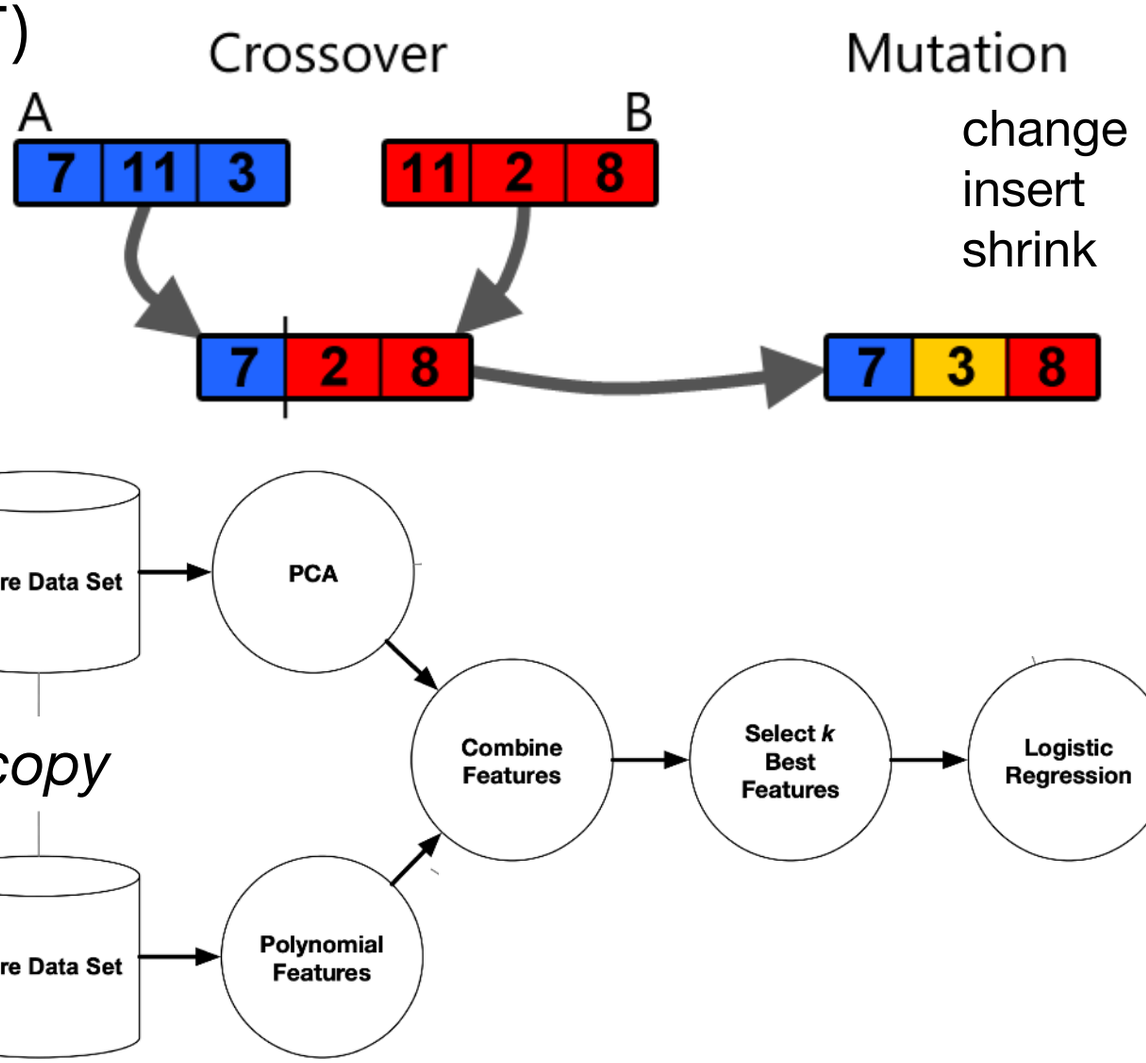
$$\Lambda_{\mathscr{A}} = \Lambda^{(1)} \times \Lambda^{(2)} \times \ldots \times \Lambda^{(m)} \times \lambda_r$$

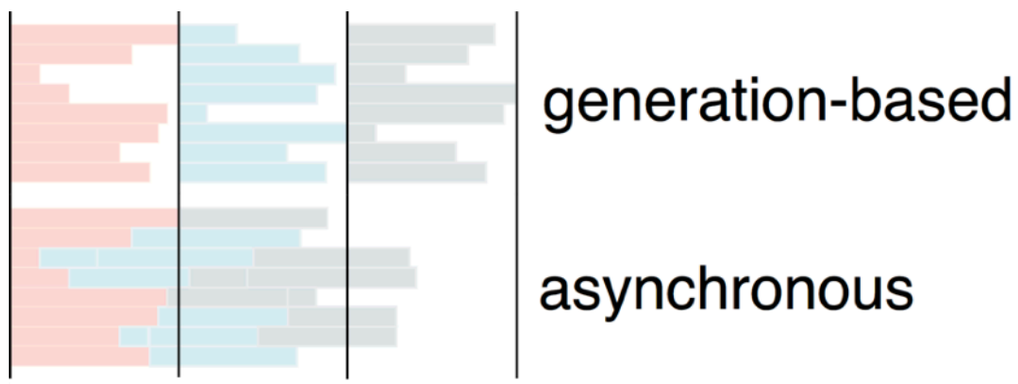


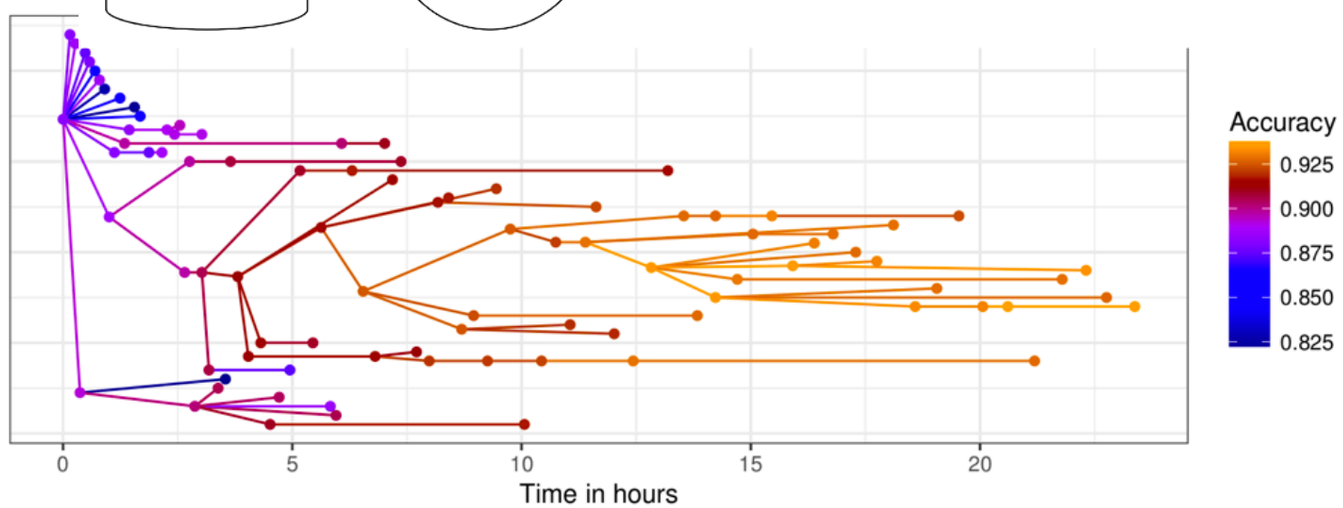
Could such a pipeline be learned?

Pipeline search: genetic programming

- Tree-based pipeline optimization (TPOT)
  - Start with random pipelines, best of every generation will cross-over or mutate
  - GP primitives include data copy and feature joins: trees
  - Multi-objective optimization: accurate but short
  - Easy to parallelize
- Asynchronous evolution (GAMA)







Pipeline search: TPOT demo

```
from tpot import TPOTClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target,
                                                    train_size=0.75, test_size=0.25)

tpot = TPOTClassifier(generations=5, population_size=50, verbosity=2, n_jobs=-1)
tpot.fit(X_train, y_train)
```

```
Optimization Progress:   0%|          | 0/300 [00:00<?, ?pipeline/s]
```

```
print(tpot.score(X_test, y_test))
```

Pipeline search: TPOT demo

```
from tpot import TPOTClassifier
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
X_train, X_test, y_train, y_test = train_test_split(digits.data, digits.target,
                                                    train_size=0.75, test_size=0.25)

tpot = TPOTClassifier(generations=5, population_size=50, verbosity=2, n_jobs=-1)
tpot.fit(X_train, y_train)
```

```
Optimization Progress:   0%|          | 0/300 [00:00<?, ?pipeline/s]
```

```
print(tpot.score(X_test, y_test))
```

Pipeline search: genetic programming

- Grammar-based genetic programming (RECIPE) *[de Sa et al. 2017]*
  - Encodes background knowledge, avoids invalid pipelines
  - Cross-over and mutation respect the grammar

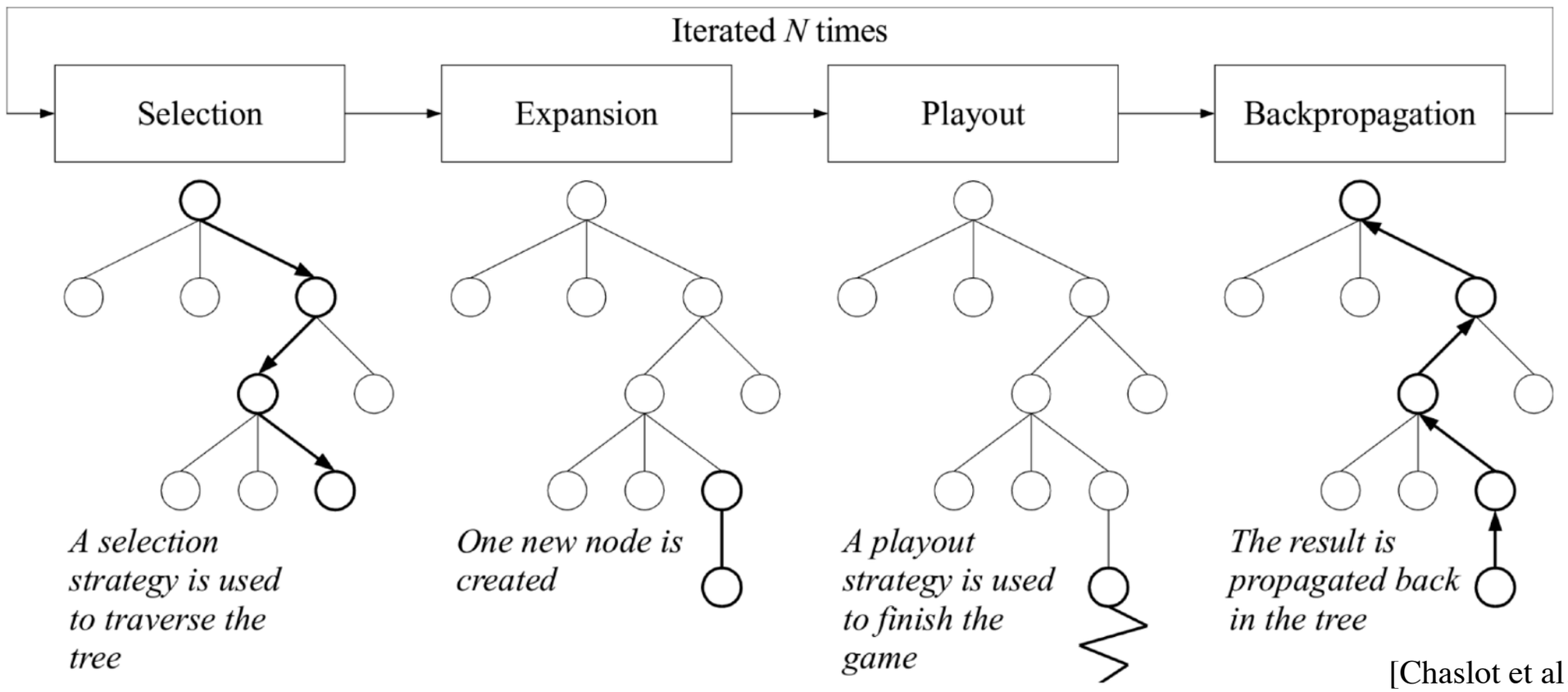
production rule *optional* *non-terminal*

```
<Start> ::= [<Pre-processing>] <Algorithm>
<Pre-processing> ::= [<Imputation>] <DimensionalityDefinition>
<Imputation> ::= Mean | Median | Max
<DimensionalityDefinition> ::= <FeatureSelection>   [<FeatureConstruction>]
                               [<FeatureSelection>] <FeatureConstruction>
<FeatureSelection> ::= <Supervised> | <Unsupervised>
<Supervised> ::= SelectKBest <K> <score> | VarianceThreshold | [...]
<score> ::= f-classification | chi2
<K> ::= 1 | 2 | 3 | [...] | NumberOfFeatures - 1
<perc> ::= 1 | 2 | 3 | [...] | 99
<Unsupervised> ::= PCA | FeatureAgglomeration <affinity> | [...]
<affinity> ::= Euclidian | L1 | L2 | Manhattan | Cosine
<FeatureConstruction> ::= PolynomialFeatures
<Algorithm> ::=  <NaiveBayes> | <Trees> | [...]
<NaiveBayes> ::= GaussianNB | MultinomialNB | BernoulliNB
```

terminal

Pipeline search: Monte Carlo Tree Search

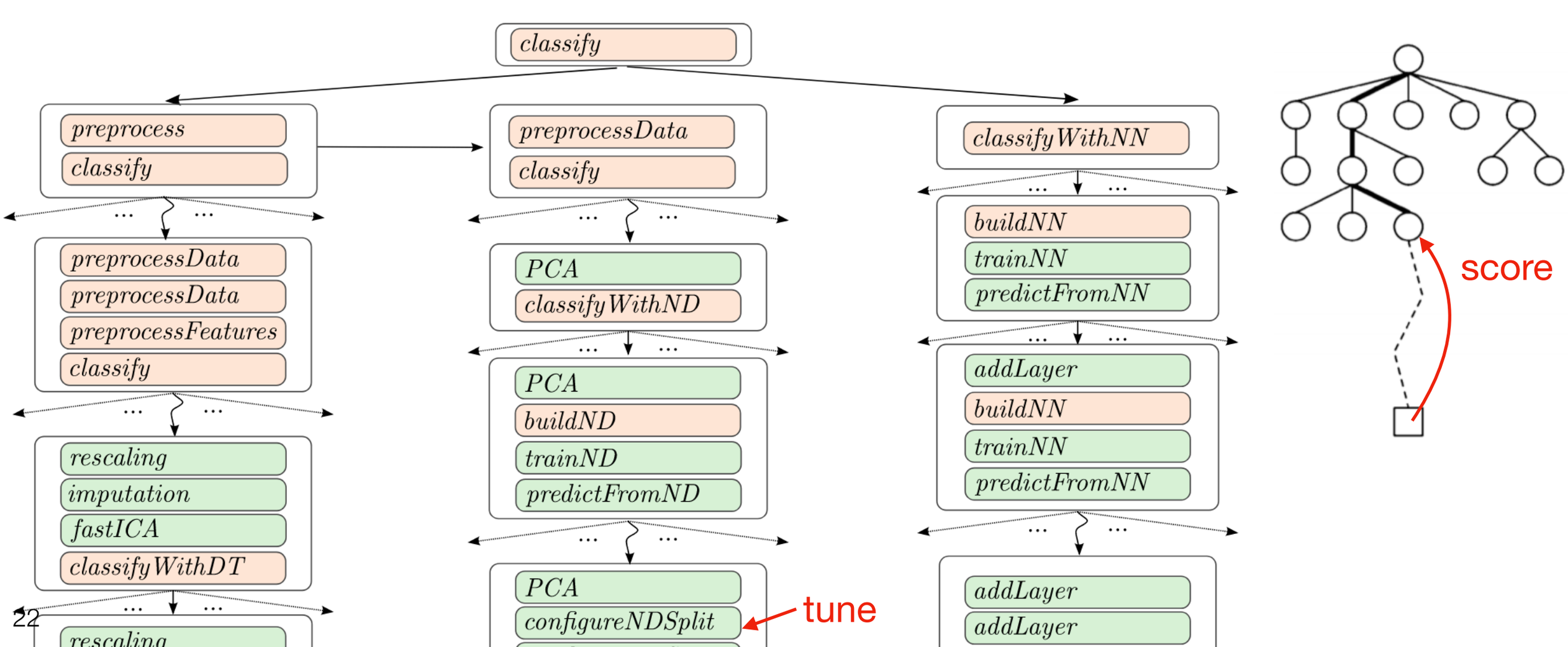
- Use MCTS to search for optimal pipelines
- Optimize the structure and hyperparameters simultaneously by building a surrogate model to predict configuration performance
  - Bayesian surrogate model: MOSAIC *[Rakotoarison et al. 2019]*
  - Neural network: AlphaD3M *[Drori et al. 2018]*



[Chaslot et al., 2008]

Pipeline search: planning

- Hierarchical planners: ML-Plan *[Mohr et al. 2018]*, *P4ML* *[Gil et al. 2018]*
  - Graph search problem, can be solved with best first search
  - Use random path completion (as in MCTS) to evaluate each node
    - Hyperparameter optimization interleaved as possible actions



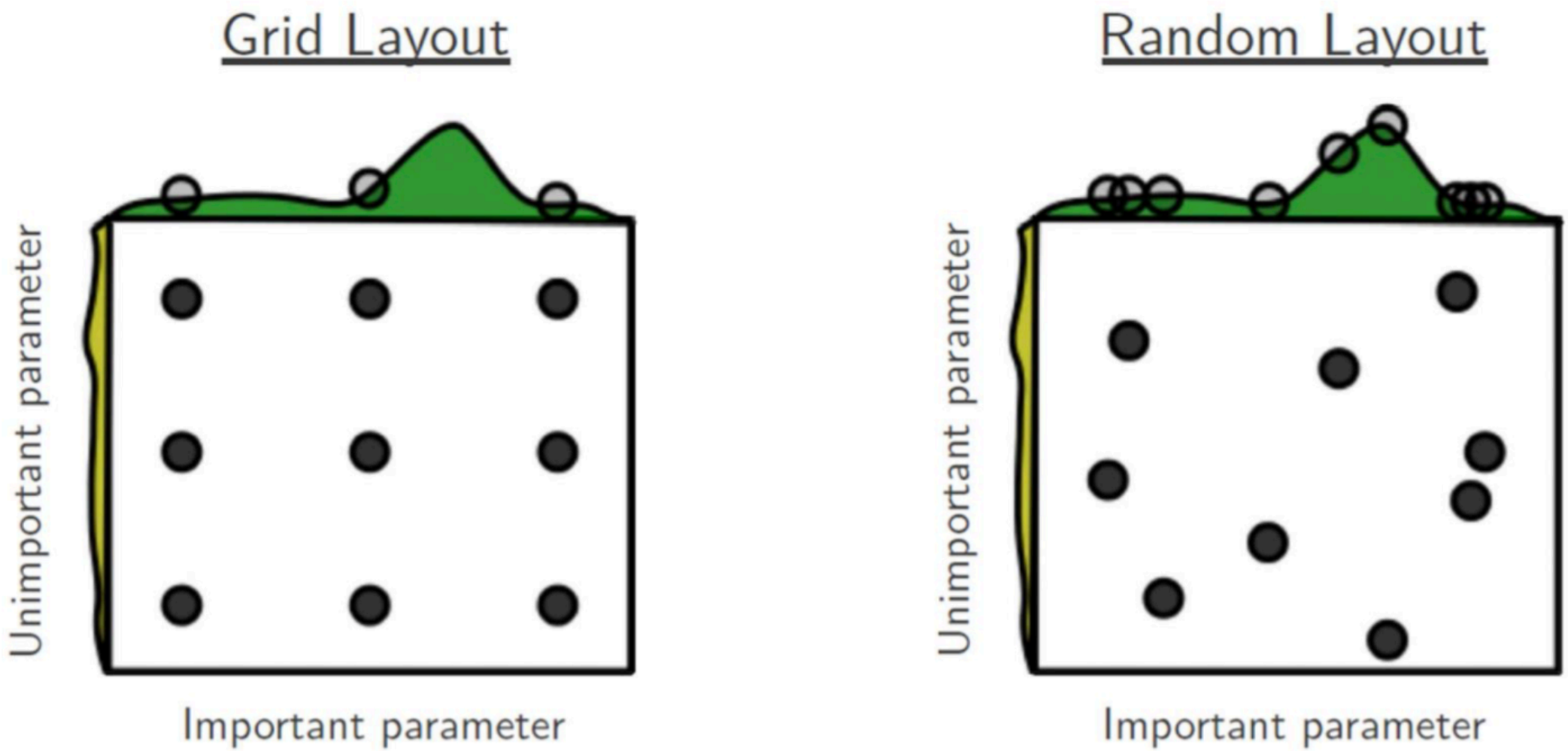
Hyperparameter optimization (HPO)

Black box optimization: expensive for machine learning.
Sample efficiency is important!

$$\boldsymbol{\lambda}^* = \underset{\lambda \in \Lambda_{\mathscr{A}}}{\operatorname{argmin}} \, \mathbb{E}_{(D_{train}, D_{valid}) \sim \mathscr{D}} \mathscr{L}(\boldsymbol{\lambda}, D_{train}, D_{test})$$

HPO: grid and random search

- Random search handles unimportant dimensions better
- Easily parallelizable, but uninformed (no learning)

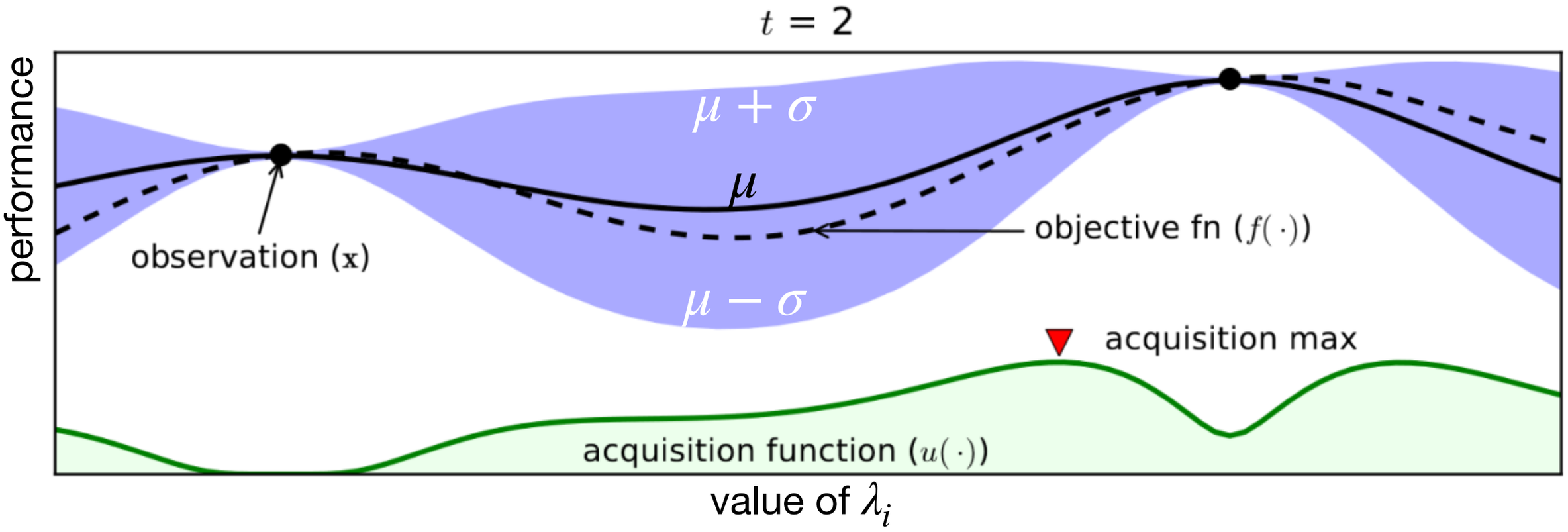


Bergstra & Bengio, JMLR 2012

Mockus [1974]

HPO: Bayesian Optimization

- Start with a few (random) hyperparameter configurations
- Build a ***surrogate model*** to predict how well other configurations will work: mean μ and standard deviation σ (blue band)
  - Any probabilistic regression model: e.g. Gaussian processes
- To avoid a greedy search, use an ***acquisition function*** to trade off exploration and exploitation, e.g. Expected Improvement (EI)
- Sample for the best configuration under that function

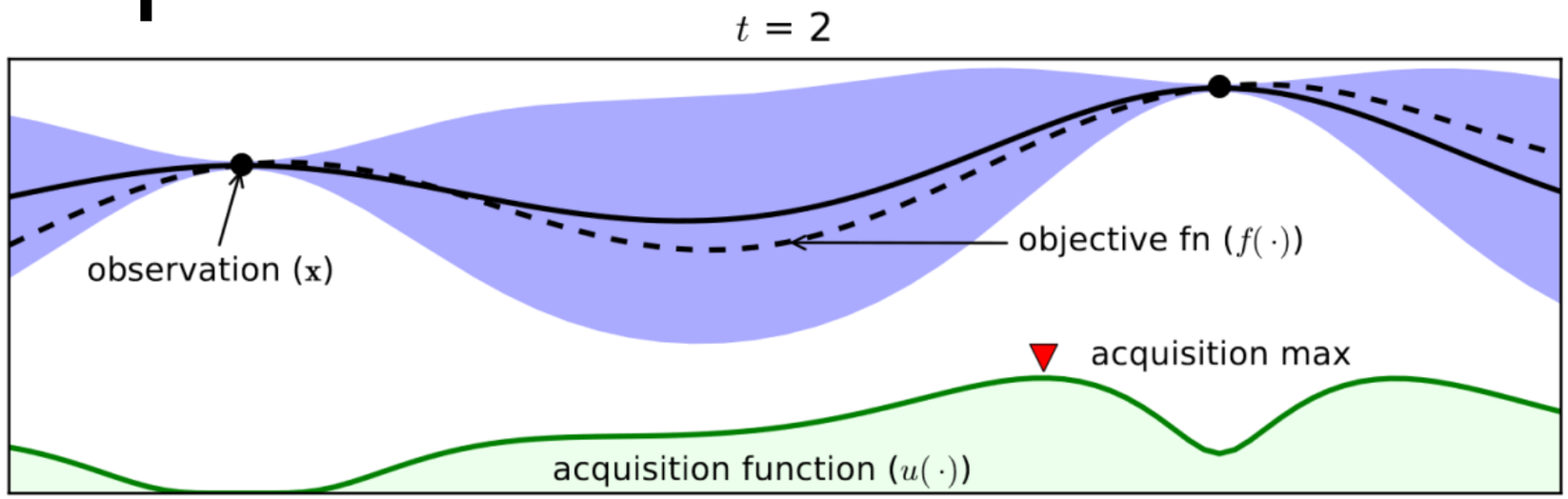


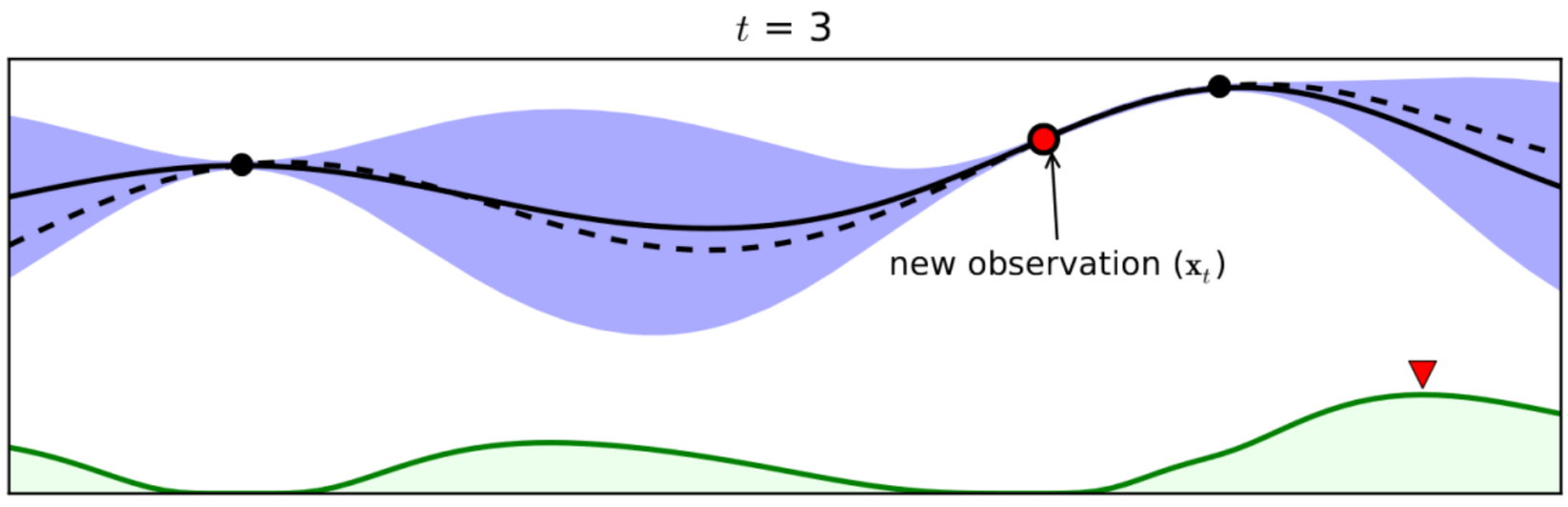
HPO: Bayesian Optimization

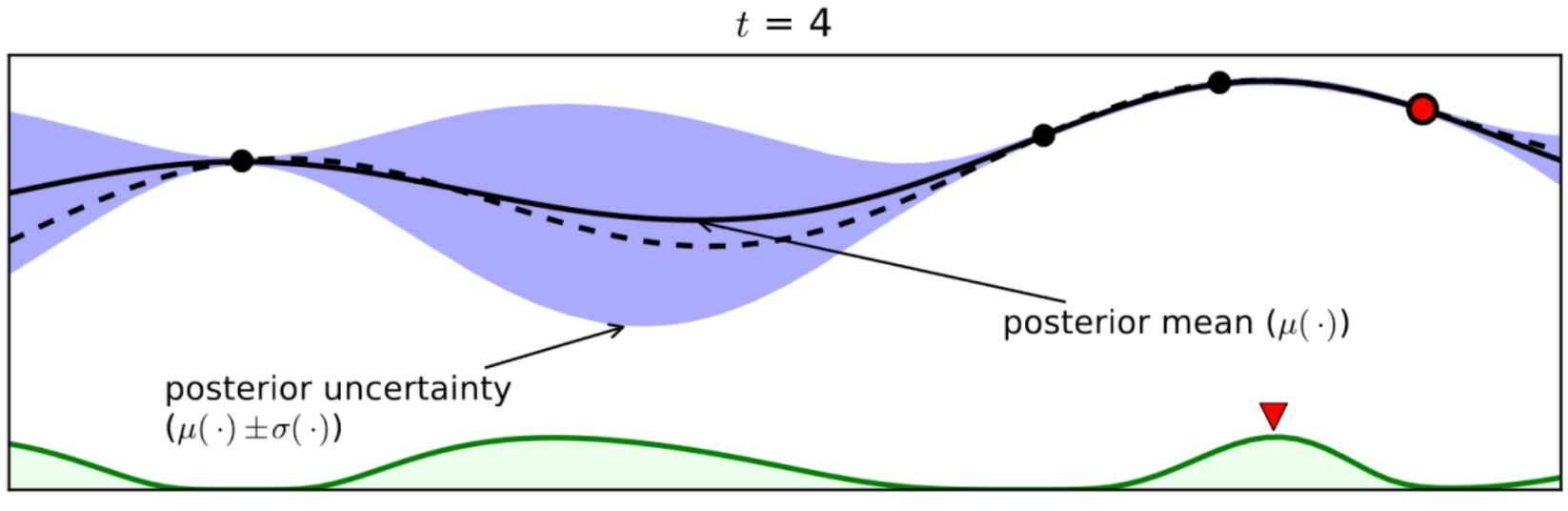
- Repeat
- Stopping criterion:
  - Fixed budget (time, evaluations)
  - Min. distance between configs
  - Threshold for acquisition function
  - Still overfits easily
- Convergence results

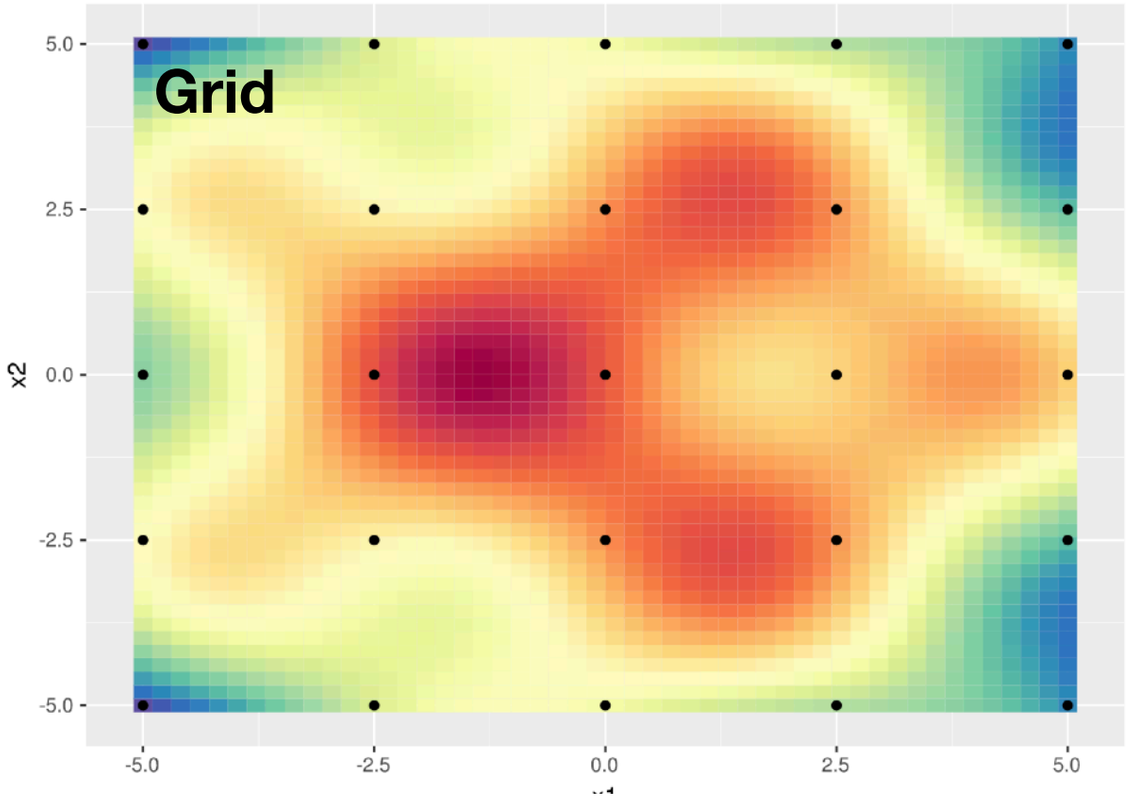
Srinivas et al. 2010, Freitas et al. 2012, Kawaguchi et al. 2016

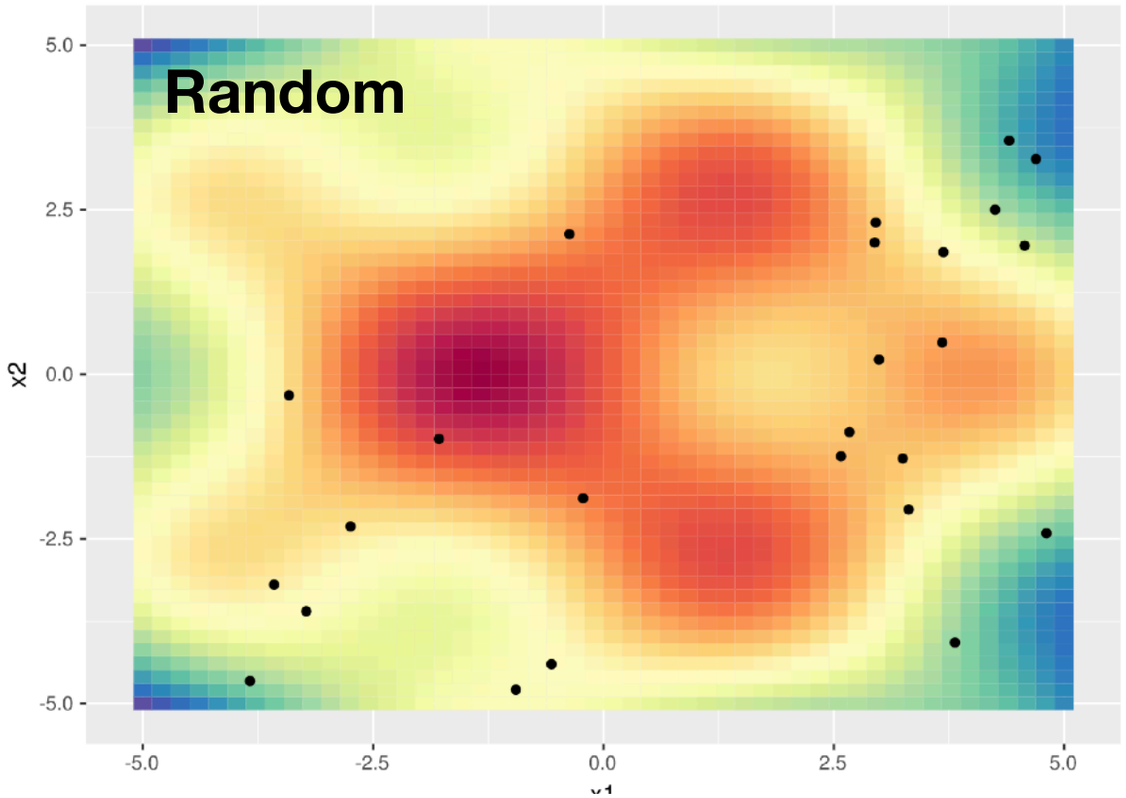
- Good for non-convex, noisy objectives
- Used in AlphaGo

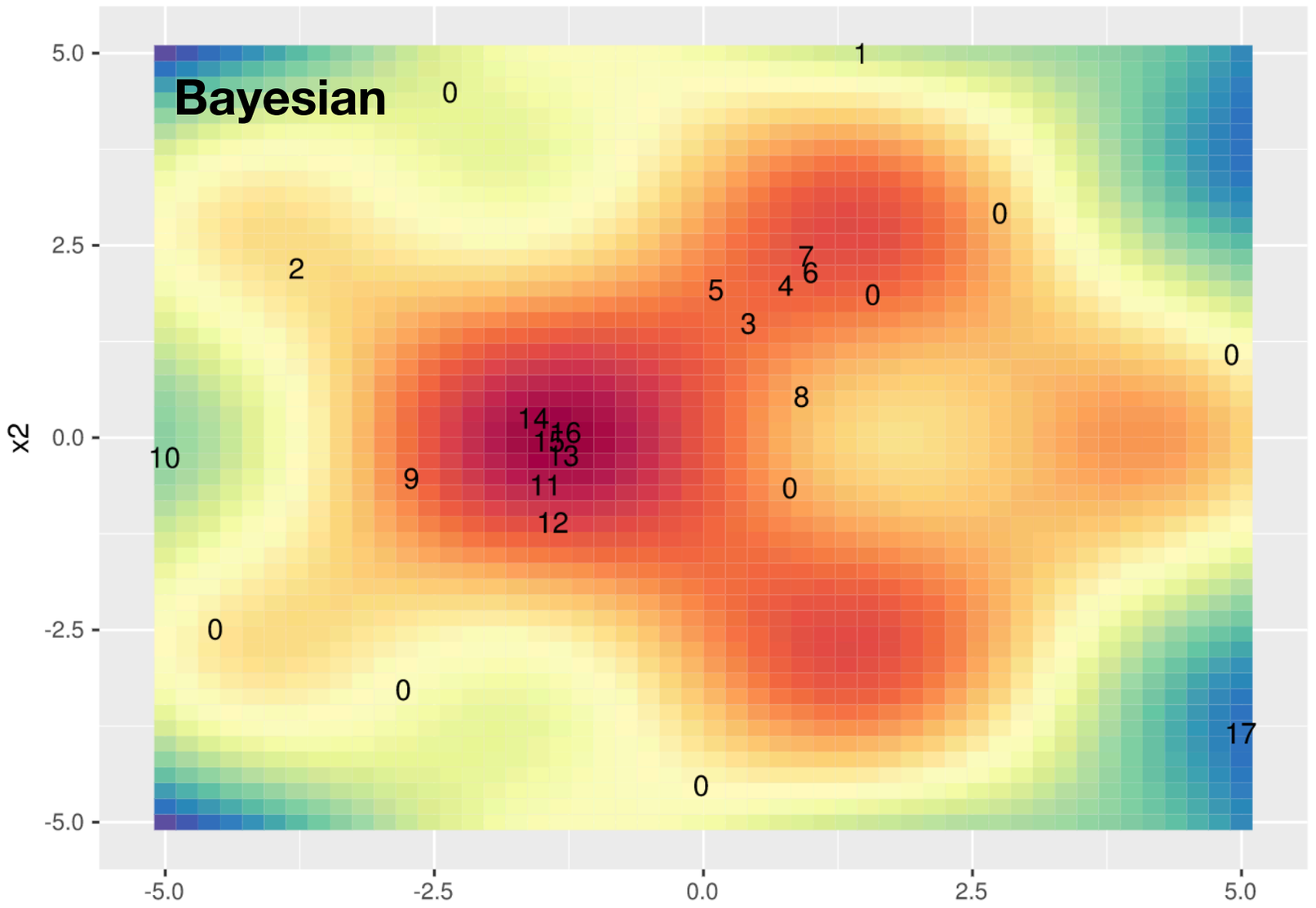







Grid
5.0
2.5
0.0
-2.5
-5.0
x2
-5.0
-2.5
0.0
2.5
5.0


Random
5.0
2.5
0.0
-2.5
-5.0
x2
-5.0
-2.5
0.0
2.5
5.0


Bayesian
5.0
2.5
0.0
-2.5
-5.0
x2
-5.0
-2.5
0.0
2.5
5.0
1
0
2
0
7
6
5
4
0
3
0
8
14
15
16
13
10
9
11
12
0
0
0
0
17

Bayesian Optimization: surrogate models

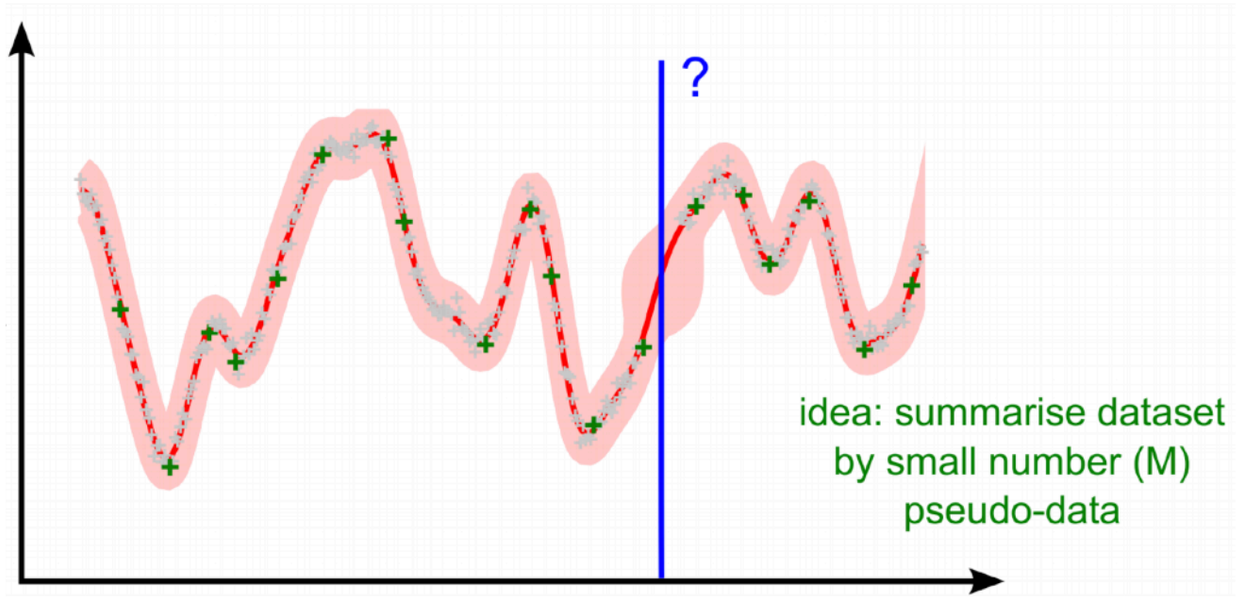
Gaussian processes

+ uncertainty, extrapolation

- Scalability (cubic)

  Sparse GPs *[Lawrence et al. 2003]*

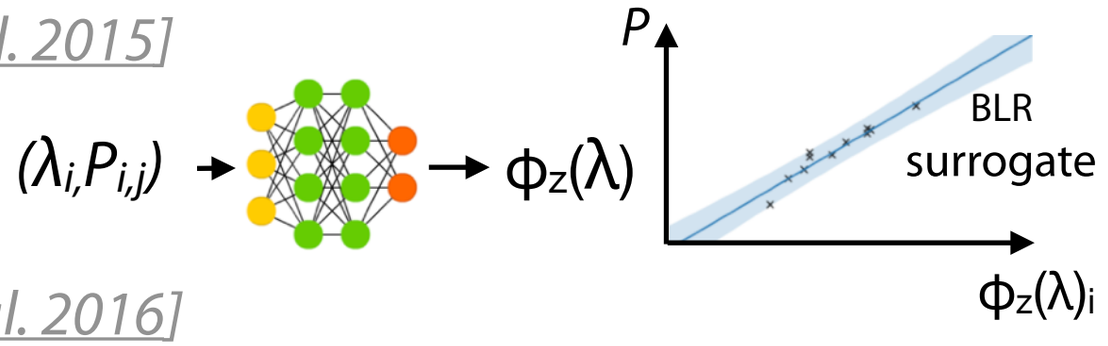
  Random embeddings *[Wang et al. 2013]*



- Robustness? Meta-BayesOpt to optimize kernel *[Malkomes et al. 2016]*

Neural networks + Bayesian LR *[Snoek et al. 2015]*

+ Scalable, Learns basis expansion Φ_z



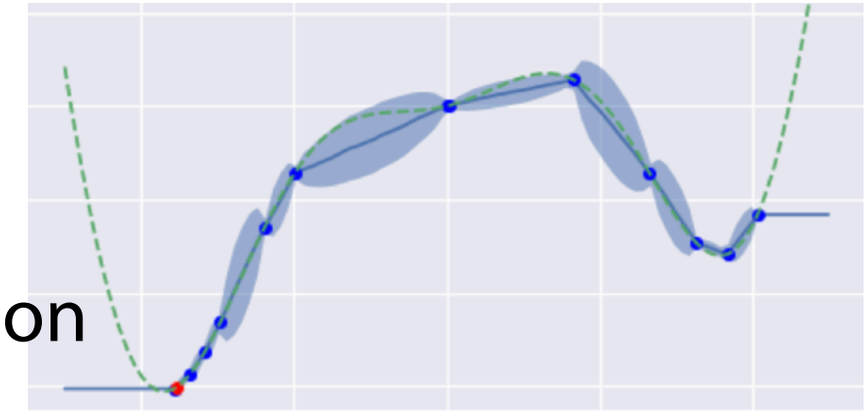
Bayesian neural networks *[Springenberg et al. 2016]*

+ Bayesian - Scalability not studied yet

Random Forests *[Hutter et al. 2011, Feurer et al. 2015]*

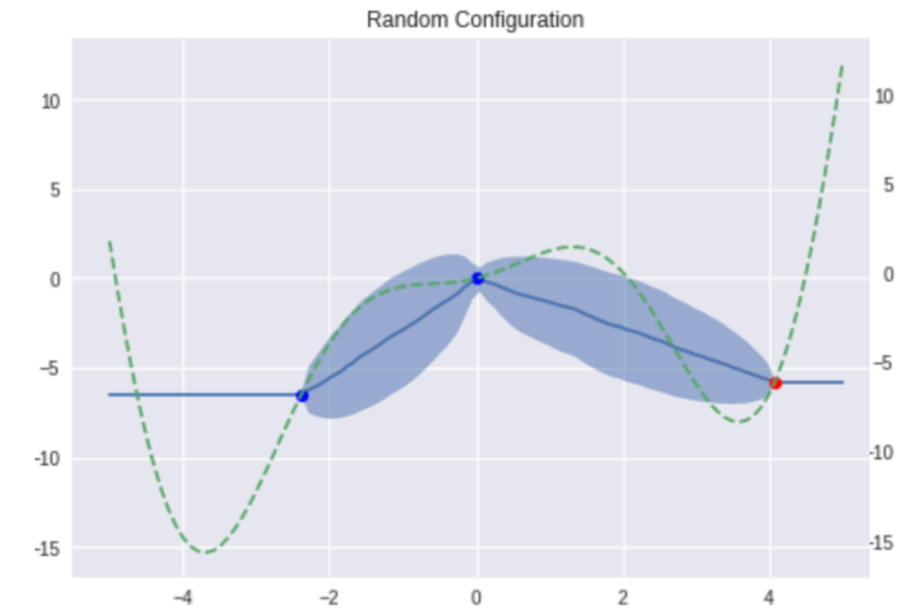
Frequentist uncertainty estimate (variance)

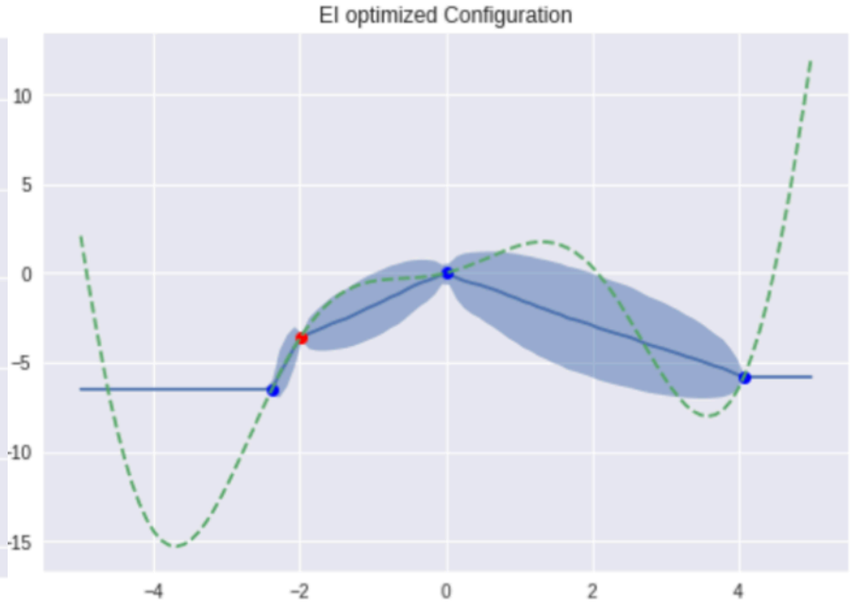
+ scalable, complex HPs - uncertainty, extrapolation

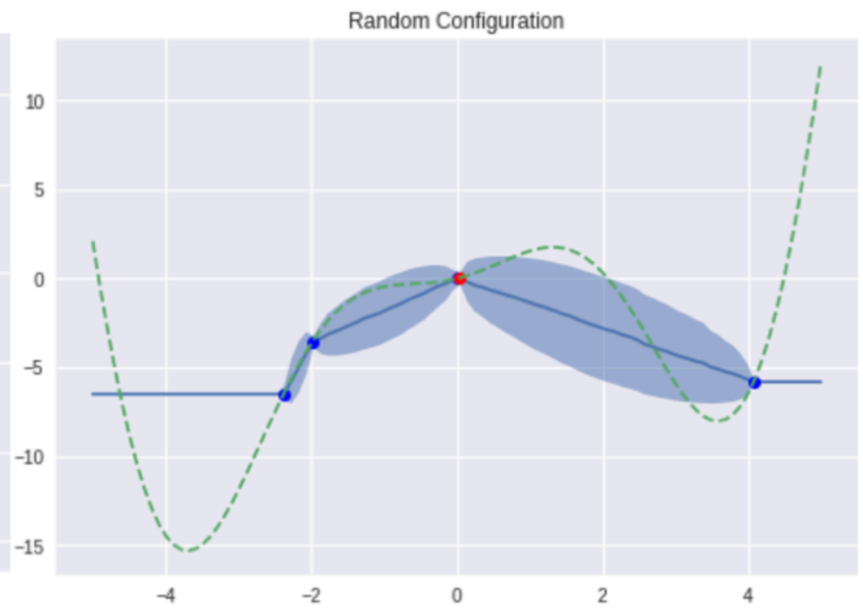


Bayesian Optimization: surrogate models

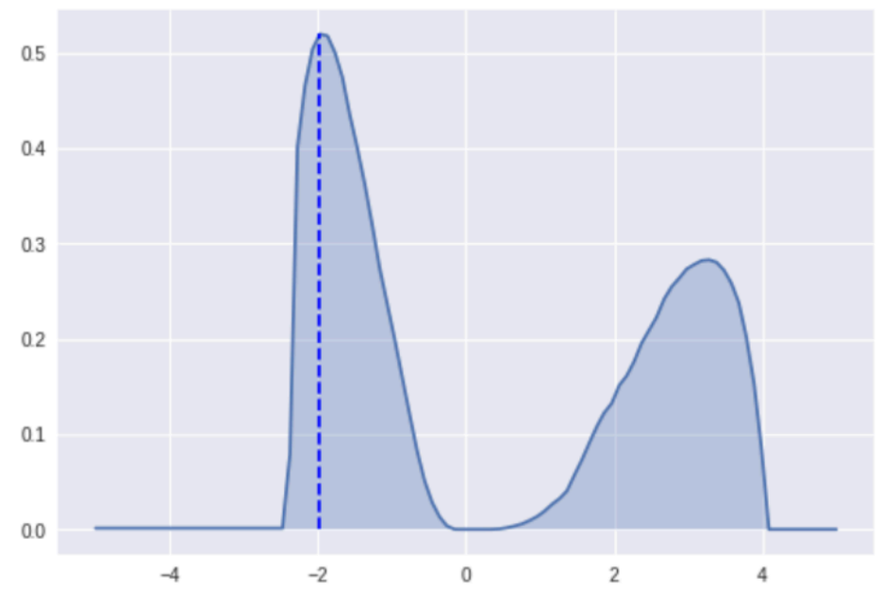
Random Forests (SMAC) example:



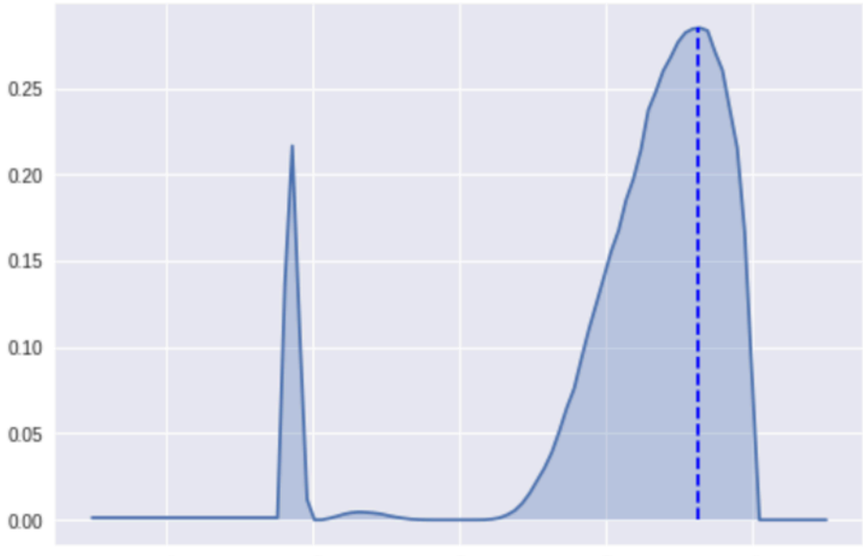




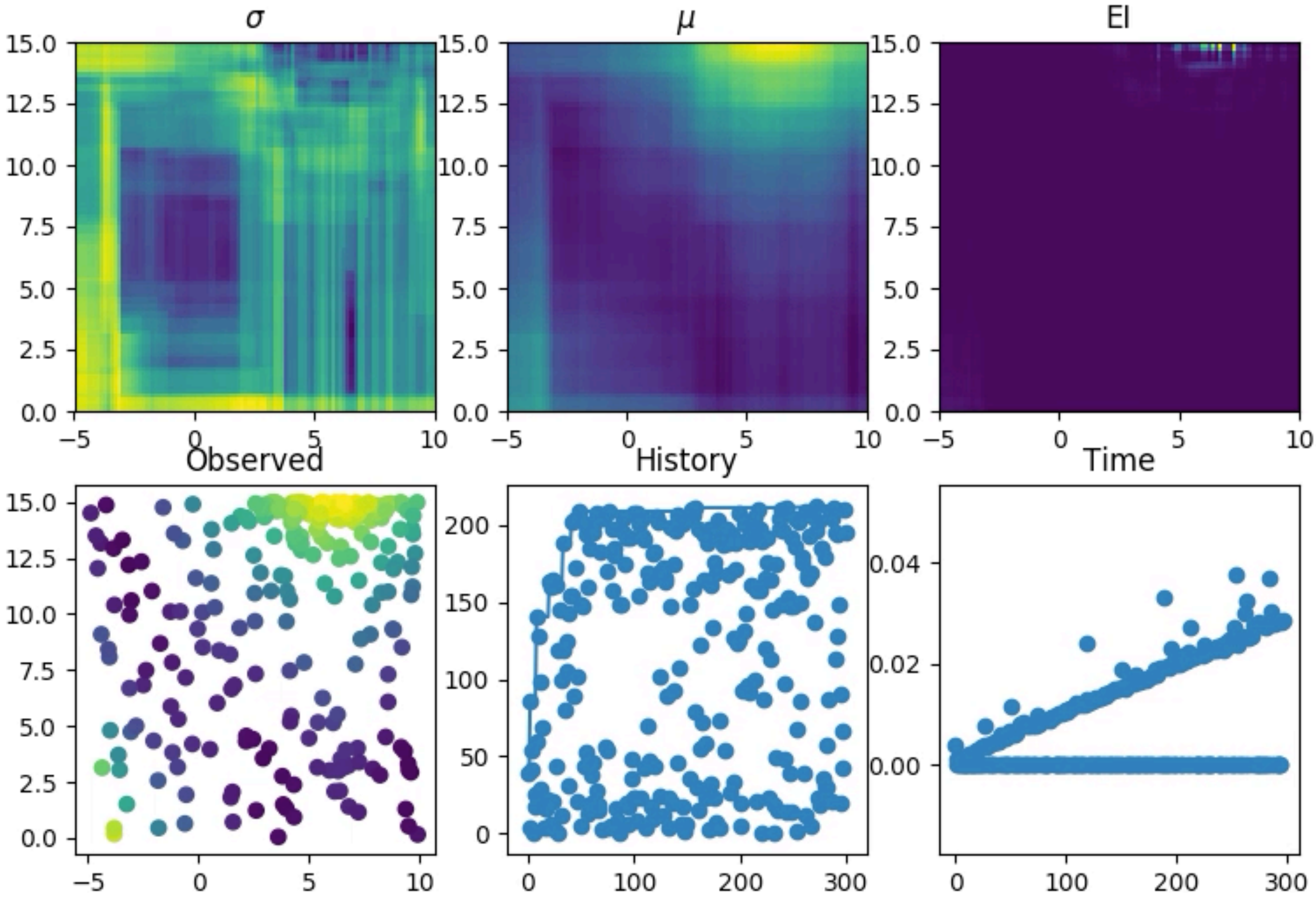
interleaved with random search



(minimizing)

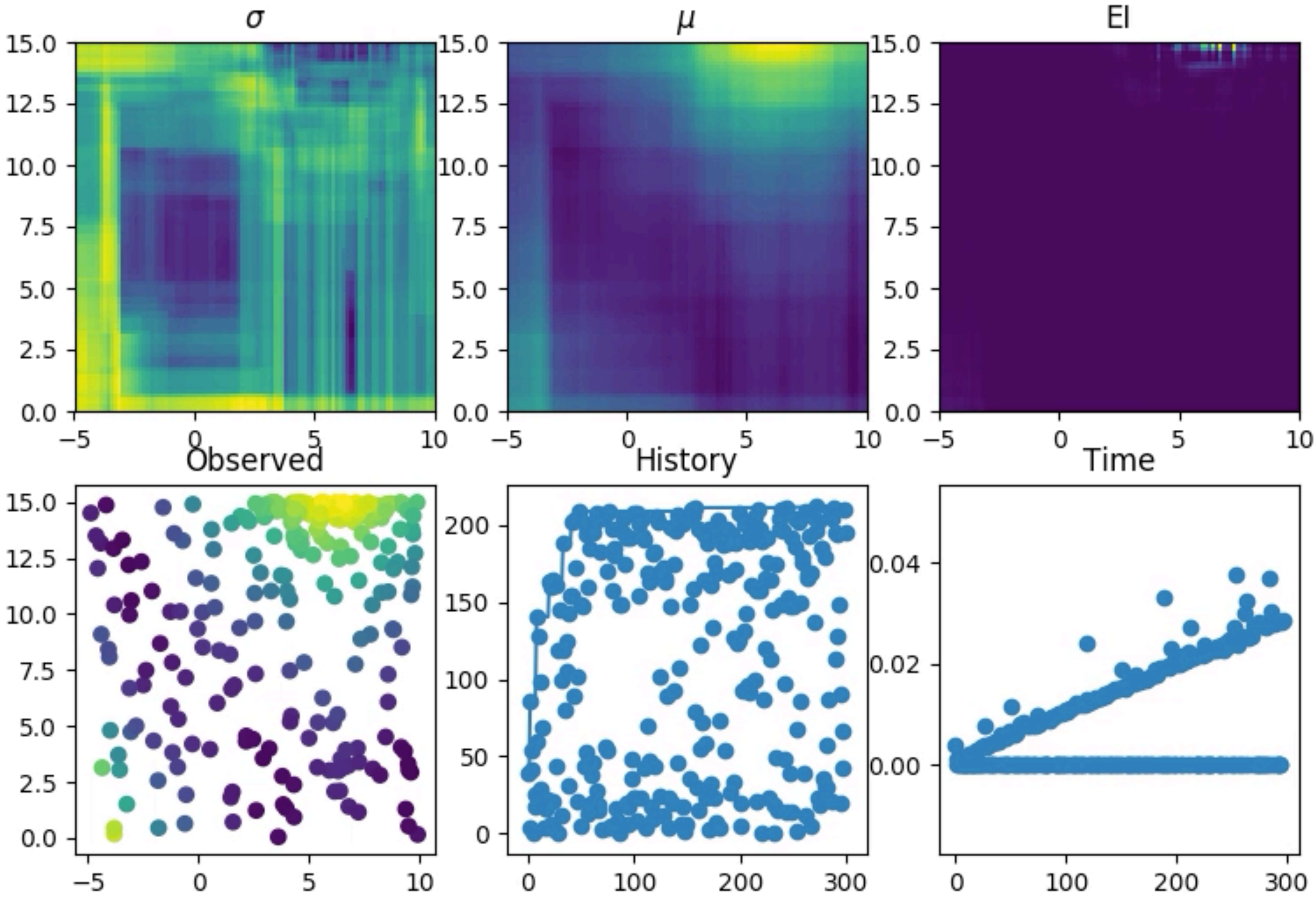


Random Forest Surrogate



animation by Jeroen van Hoof

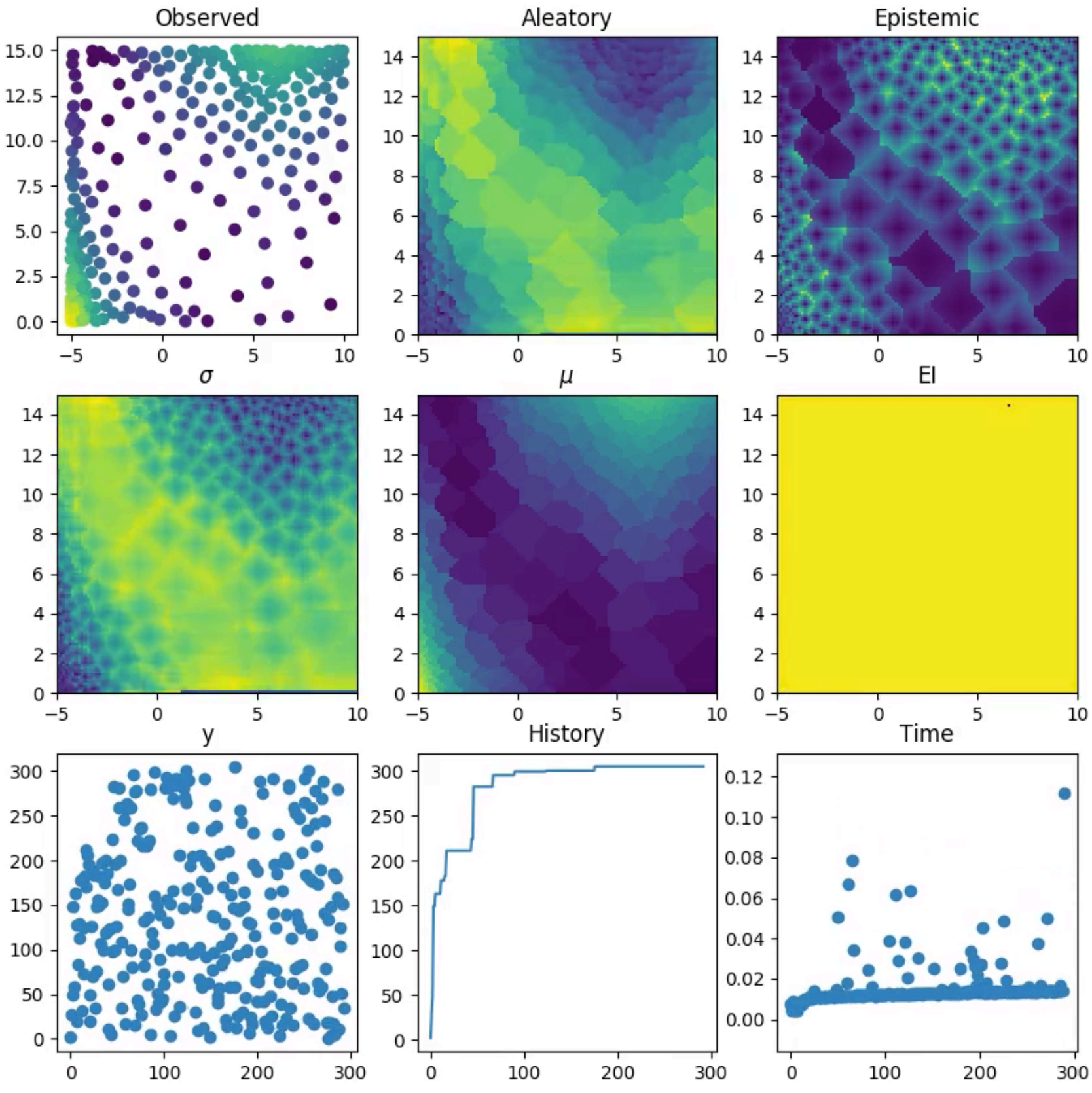
Random Forest Surrogate



animation by Jeroen van Hoof

Gradient Boosting Surrogate

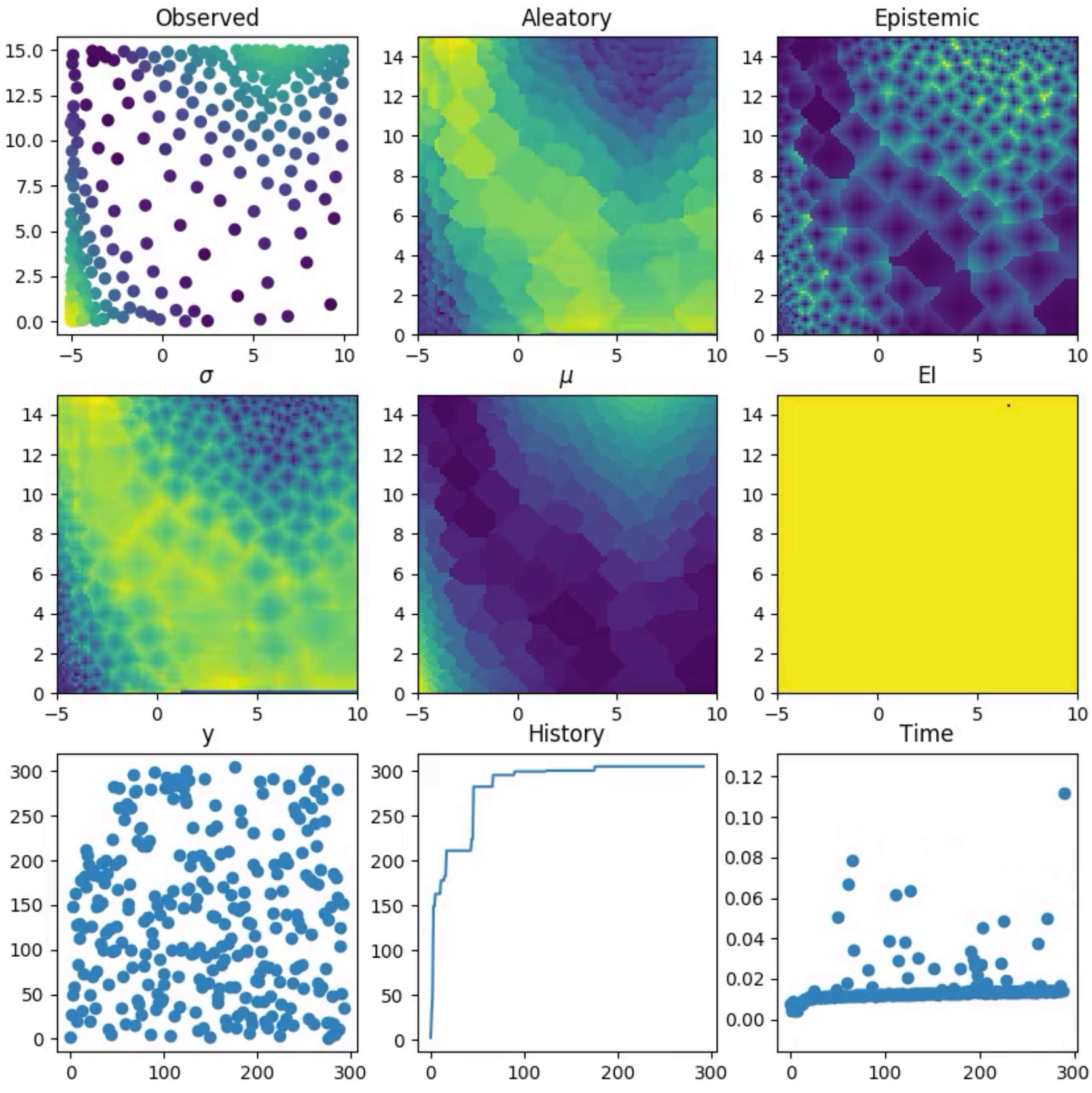
Estimate uncertainty with quantile regression



Jeroen van Hoof (under review)

Gradient Boosting Surrogate

Estimate uncertainty with quantile regression



Jeroen van Hoof (under review)

HPO: Tree of Parzen Estimators

1.Test some hyperparameters

2. Separate into good and bad hyperparameters (with some quantile)

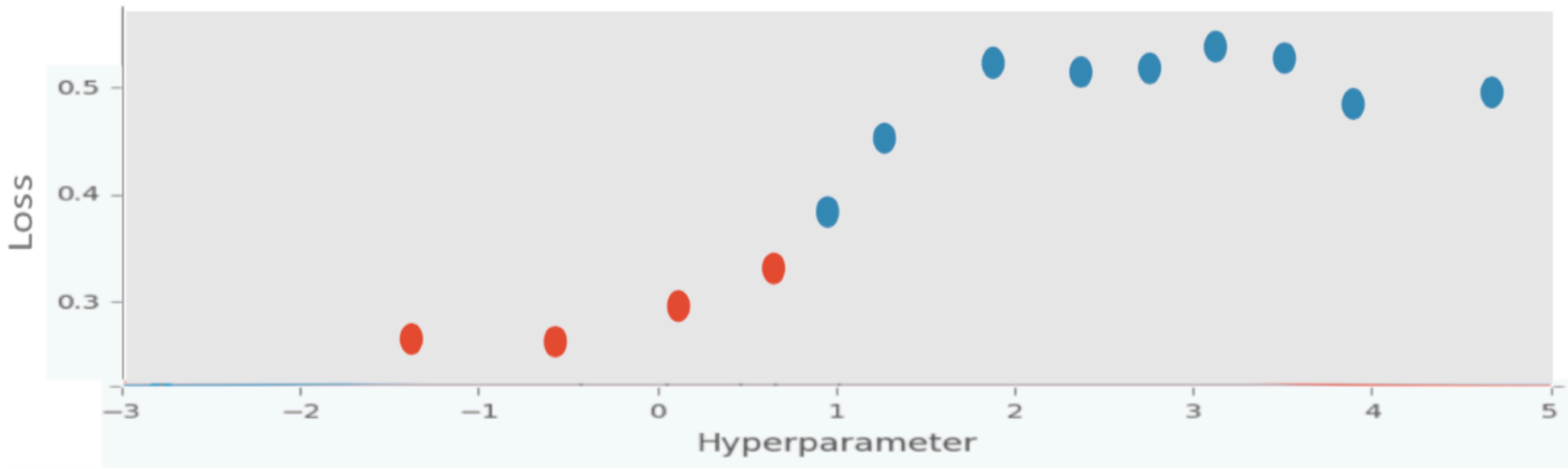
3. Fit non-parametric KDE for $p(\lambda = \text{good})$ and $p(\lambda = \text{bad})$

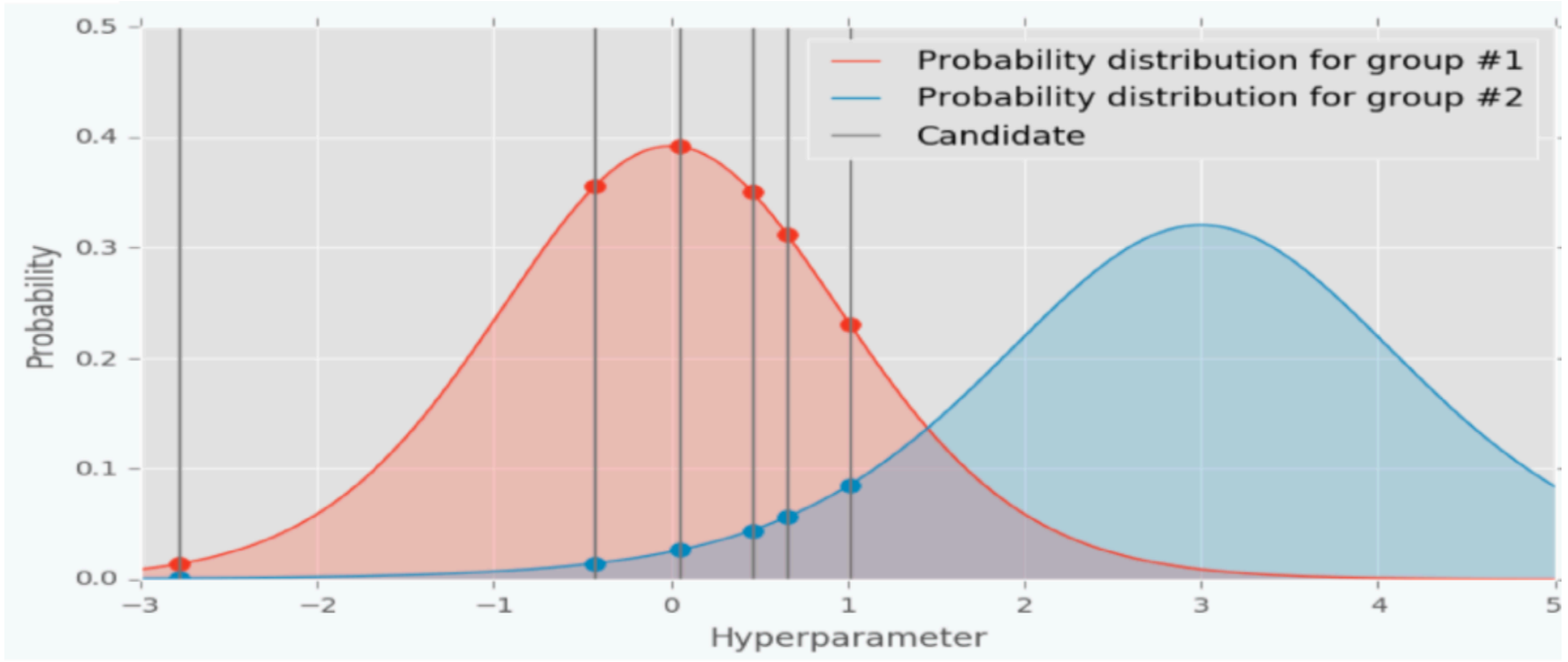
4. For a few samples, evaluate $\dfrac{p(\lambda = \text{good})}{p(\lambda = \text{bad})}$

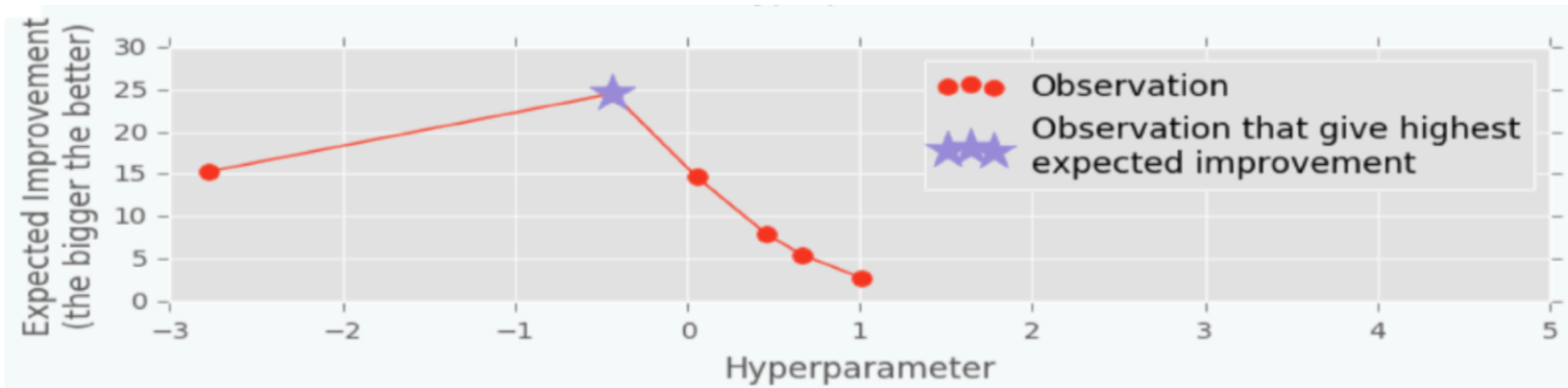
Shown to be equivalent to EI!

Efficient, parallelizable, robust, but less sample efficient than GPs

Used in HyperOpt-sklearn [Komer et al. 2019]



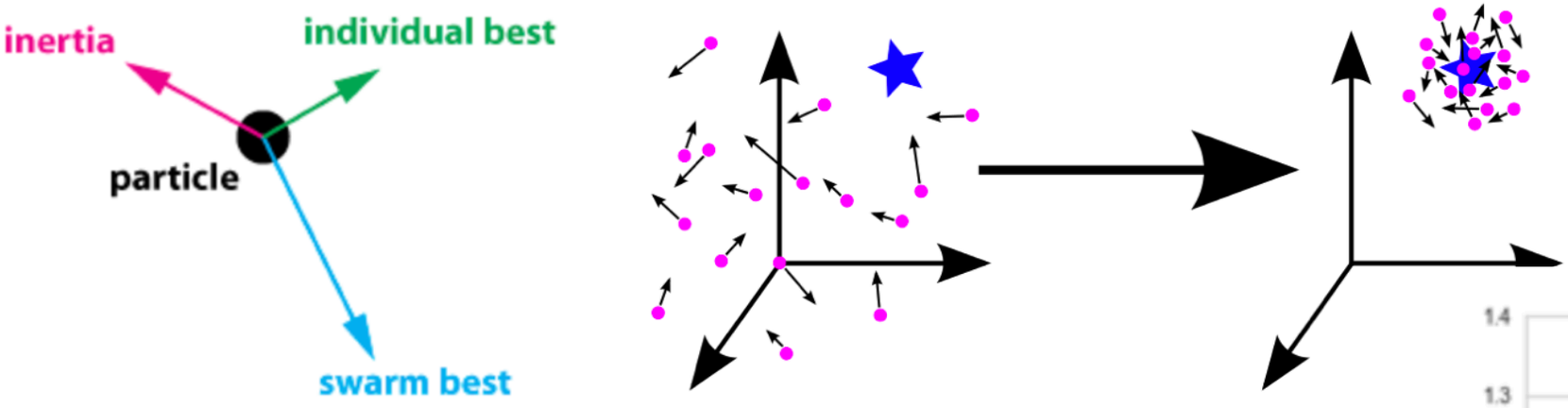




HPO: population-based methods

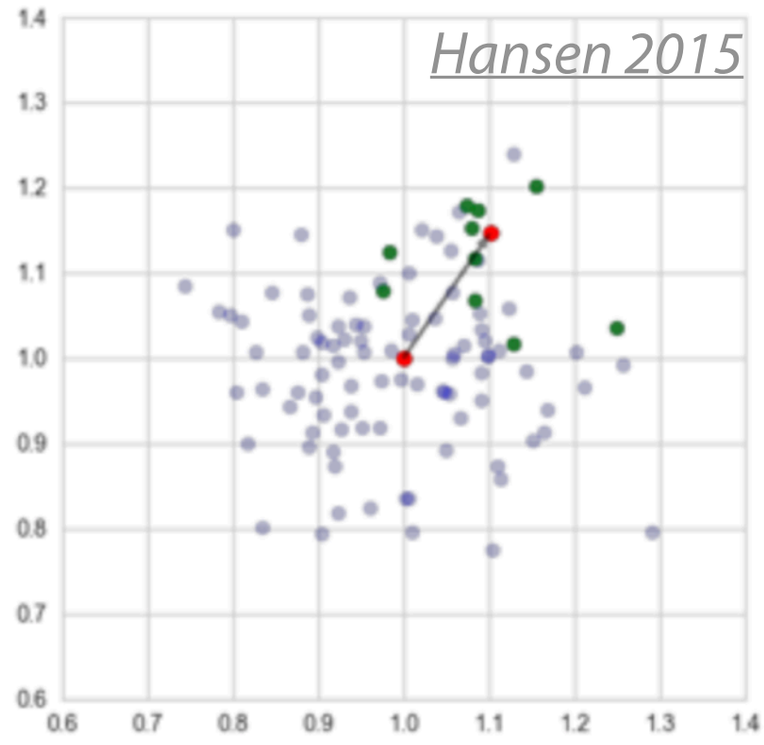
- Less sample efficient, but easy to parallelize, and adapts to changes
- Genetic programming *Olson, Moore 2016, 2019*
  - Mutations: add, mutate/tune, remove HP

- Particle swarm optimization *Mantovani et al 2015*



- Covariance matrix adaptation evolution (CMA-ES)
  - Purely continuous, expensive
  - Very competitive to optimize deep neural nets

[Loshilov, Hutter 2016]

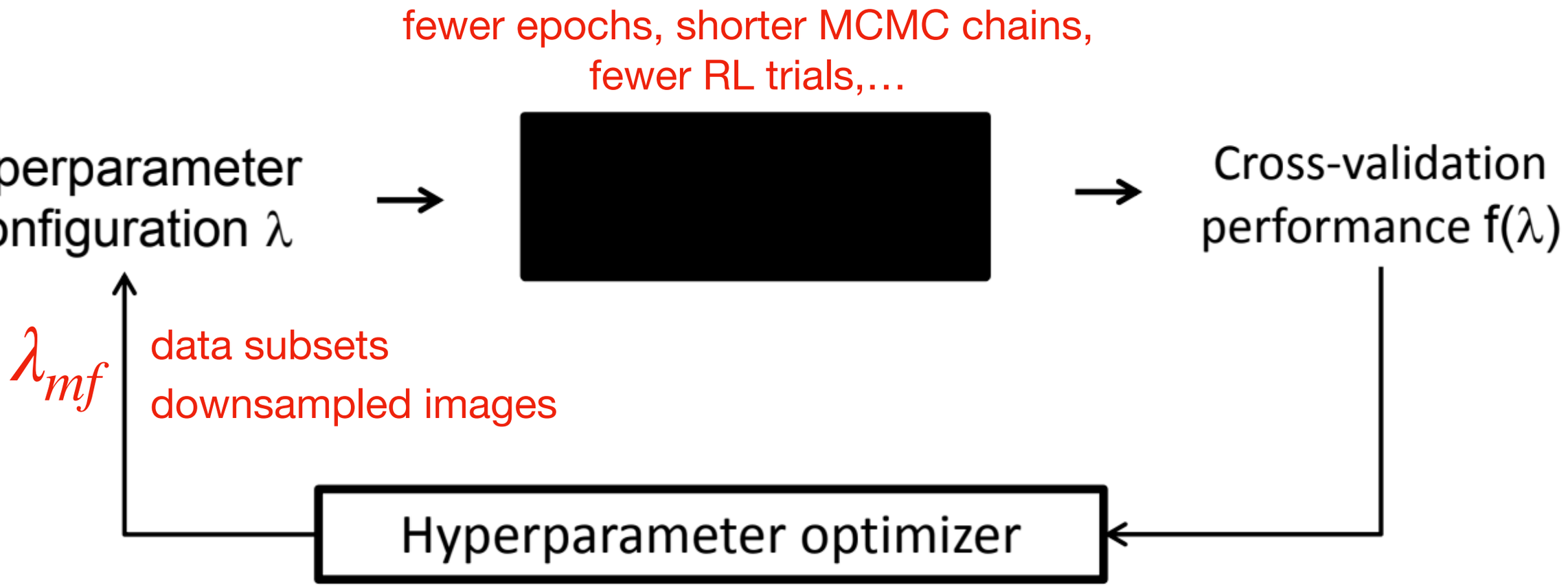


AutoML performance improvements

Making it practically useful (inspired by humans)

Multi-fidelity approaches

General techniques for cheap approximations of black-box

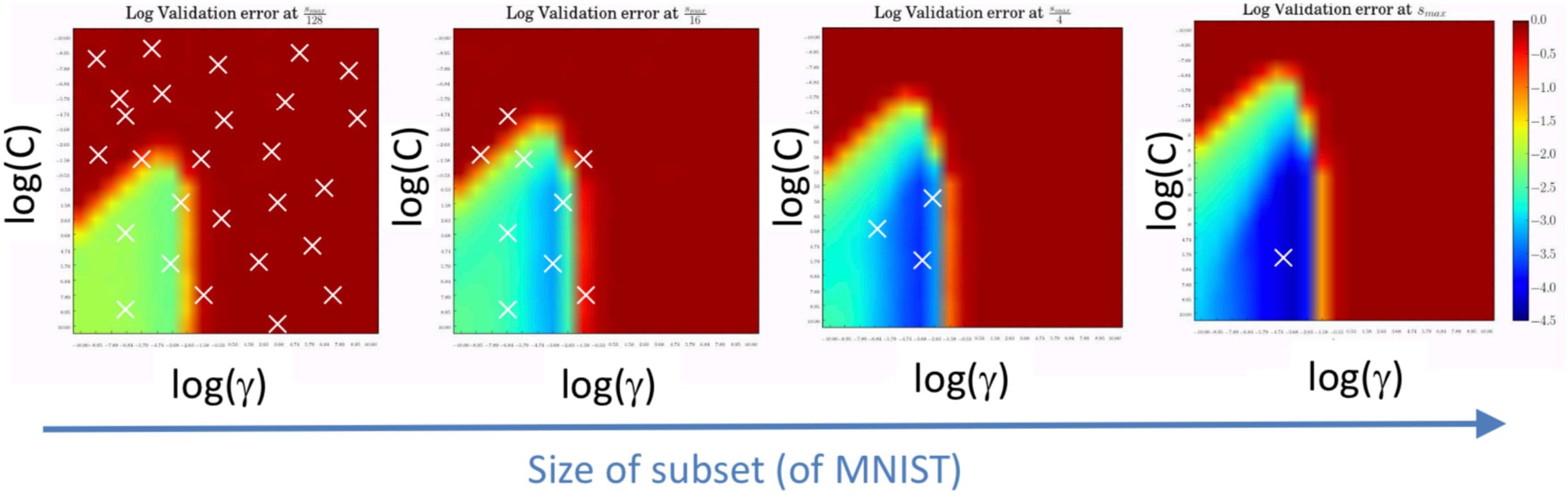


$$\lambda^* = \underset{\lambda \in \Lambda_{\mathcal{A}}}{\operatorname{argmin}} \, \mathbb{E}_{(D_{train}, D_{valid}) \sim \mathcal{D}} \mathcal{L}(\lambda, D_{train}, D_{test})$$

Multi-fidelity approaches

Cheap low-fidelity surrogates: train on small subsets, infer which regions may be interesting to evaluate in more depth

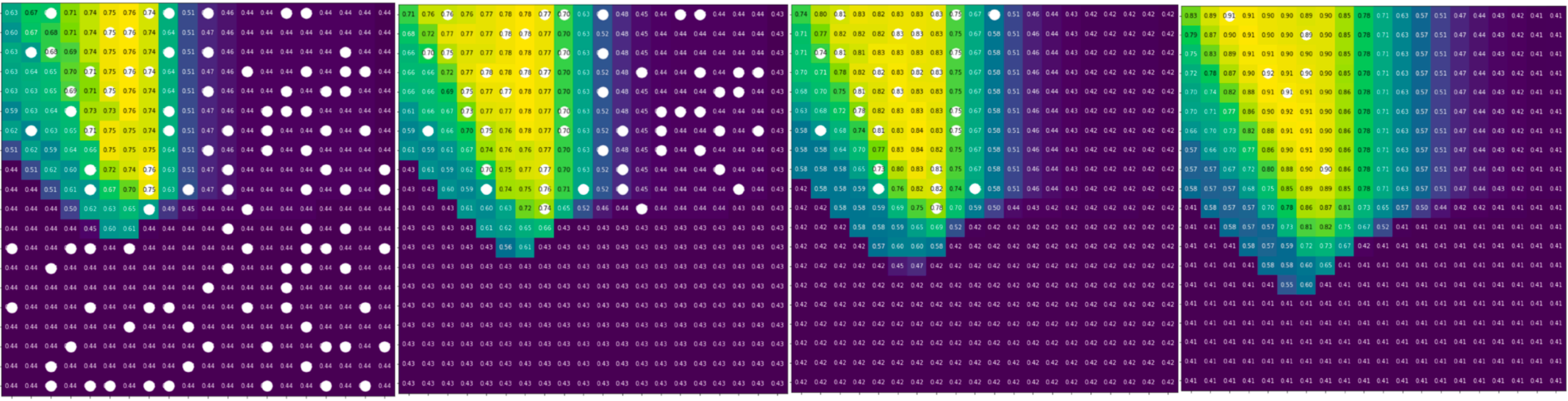
- Evaluate random samples on smallest set
- Update a Bayesian surrogate model
- Select fewer points based on surrogate, evaluate on more data, repeat



Multi-fidelity approaches

Successive halving:

- Randomly sample candidates and evaluate on a small data sample
- retrain the 50% best candidates on twice the data

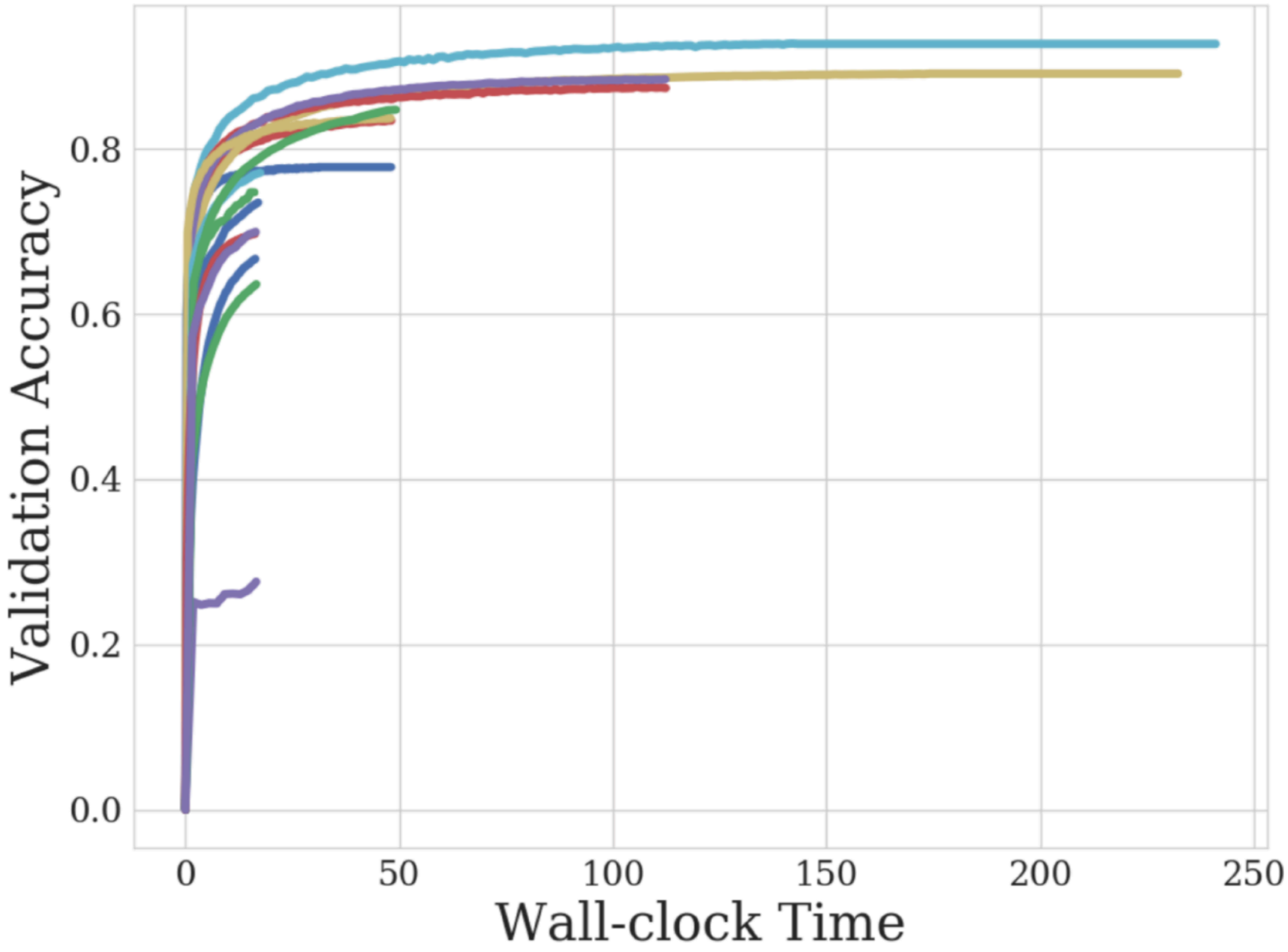


1/16 1/8 1/4 1/2

Multi-fidelity approaches

Successive halving:

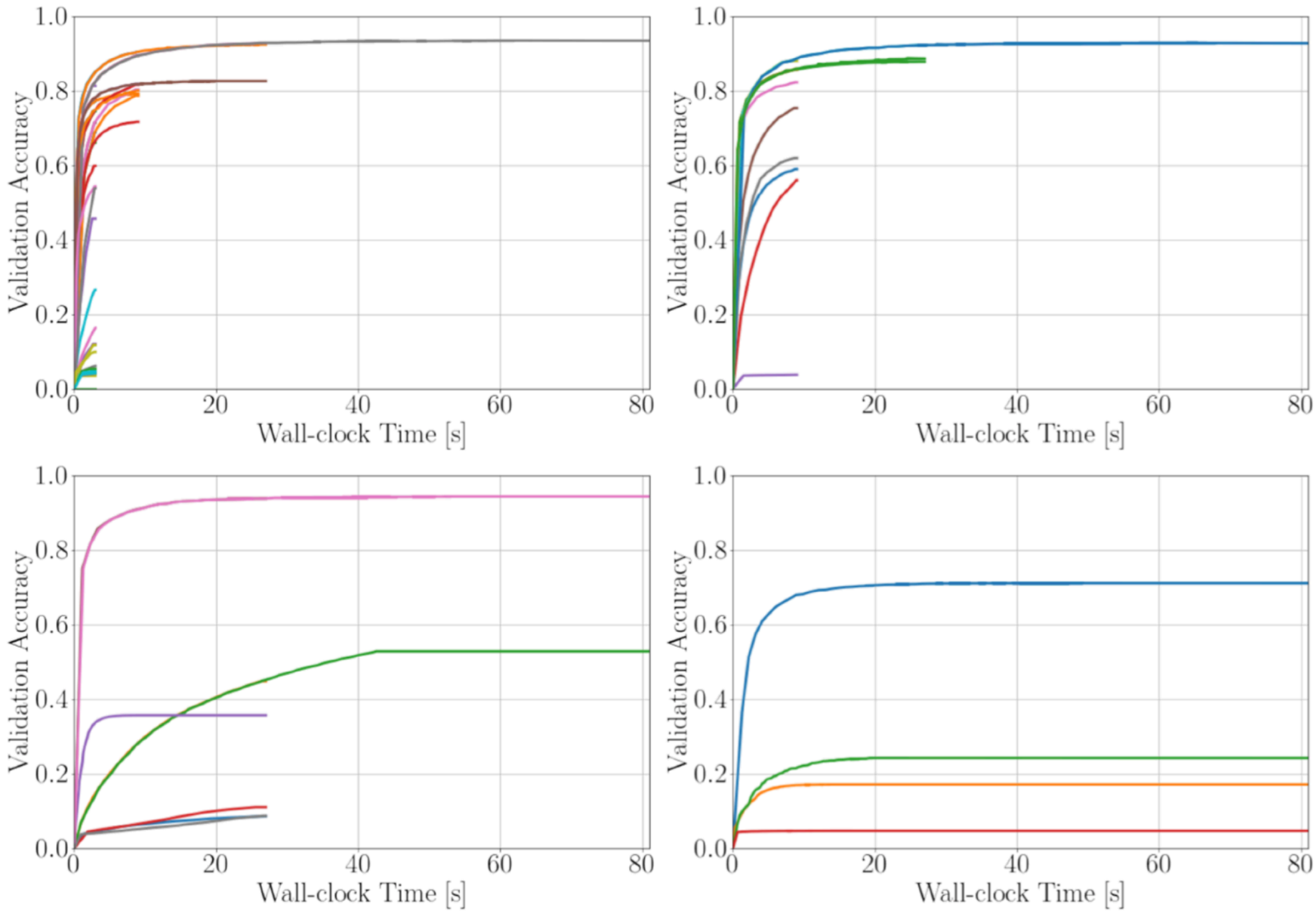
- Randomly sample candidates and evaluate on a small data sample
- retrain the best half candidates on twice the data



Multi-fidelity approaches

Hyperband: Repeated, decreasingly aggressive successive halving

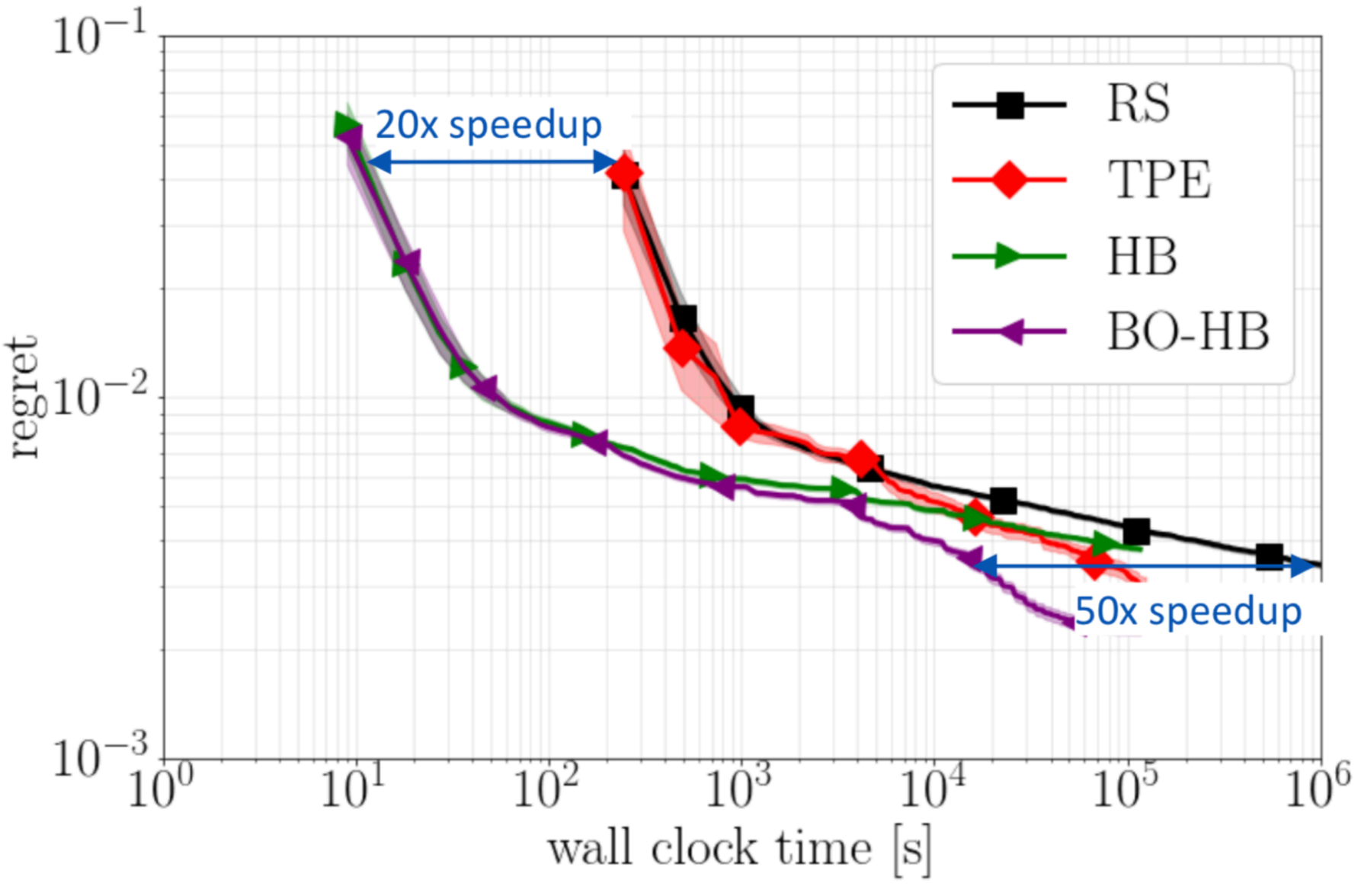
- Minimizes (doesn't eliminate) chance that candidate was pruned to early
- Strong anytime performance, easy to implement, scalable, parallelizable



Multi-fidelity approaches

Combined Bayesian Optimization and Hyperband (BO-HB)

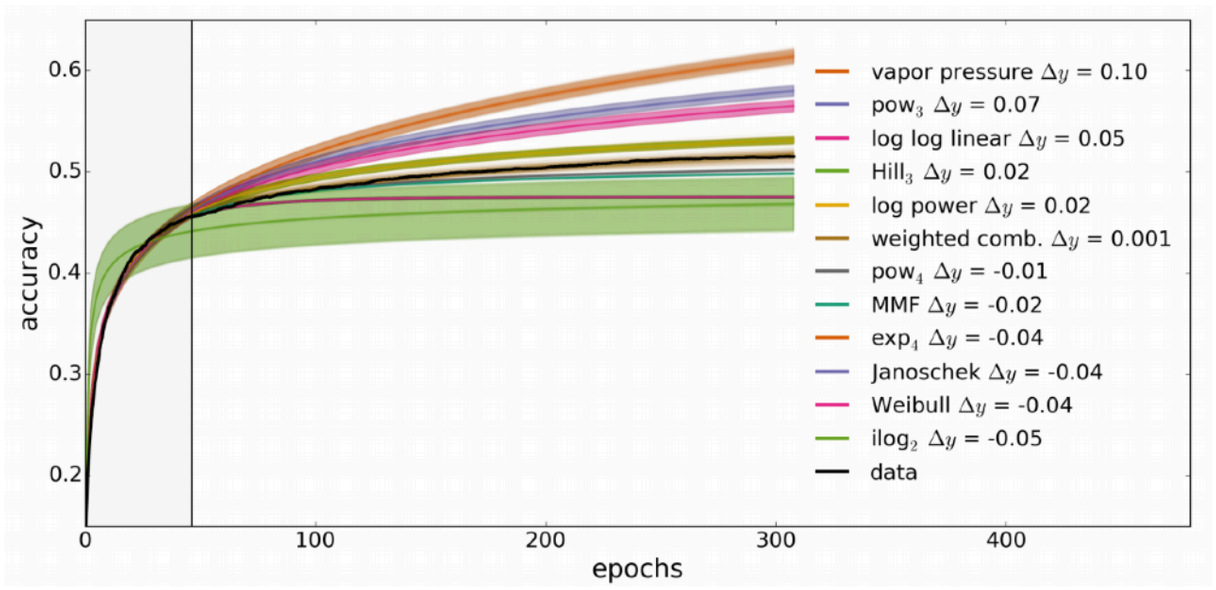
- BayesOpt: choose which configurations to evaluate
- Hyperband: allocate budgets more efficiently
- Strong anytime and final performance

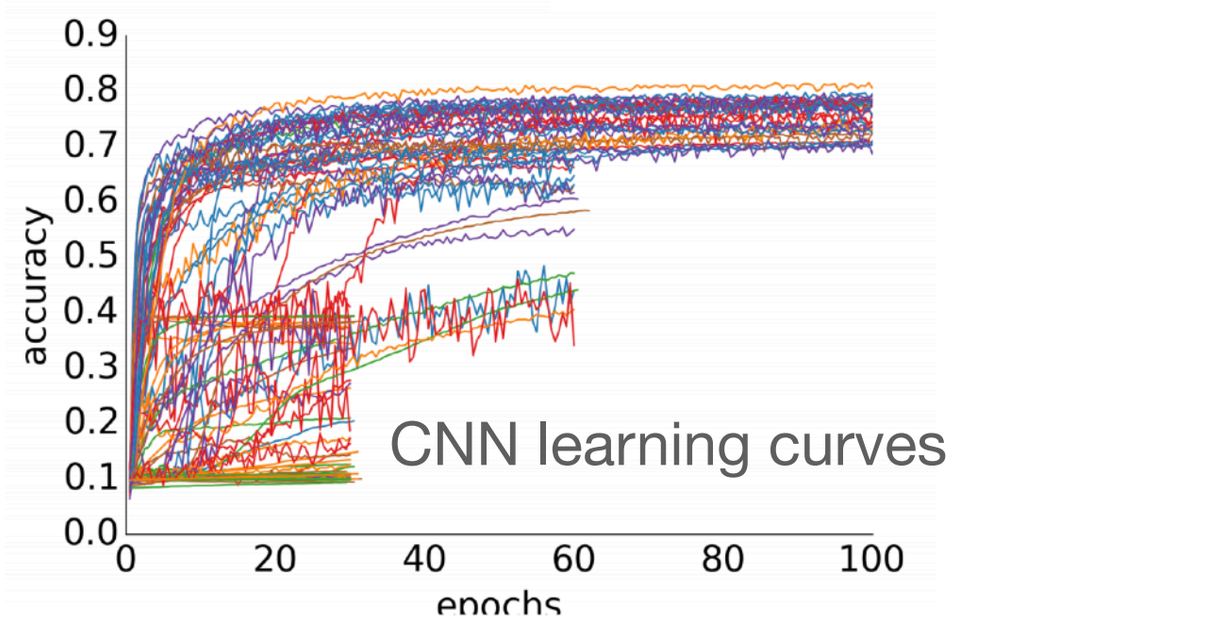


Early stopping

Learning curve prediction

- Model learning curves as parametric functions, fit *[Domhan et al 2015]*
- Train a Bayesian Neural Net to predict learning curves *[Klein et al 2017]*

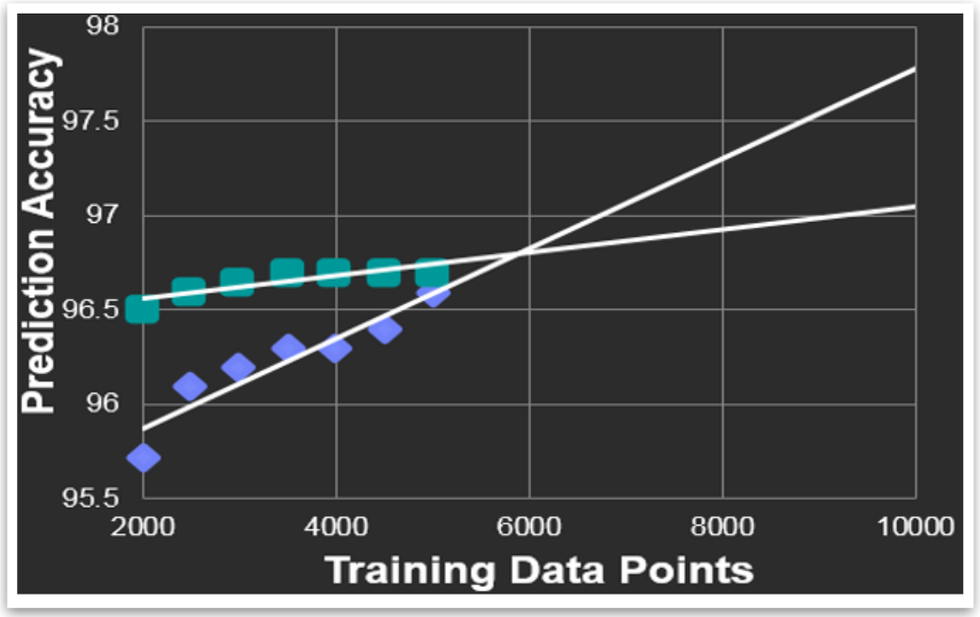




Data allocation using upper bounds (DAUB)

- Fit linear function through latest estimates
- Compute upper performance bound

Sabharwal et al. 2015



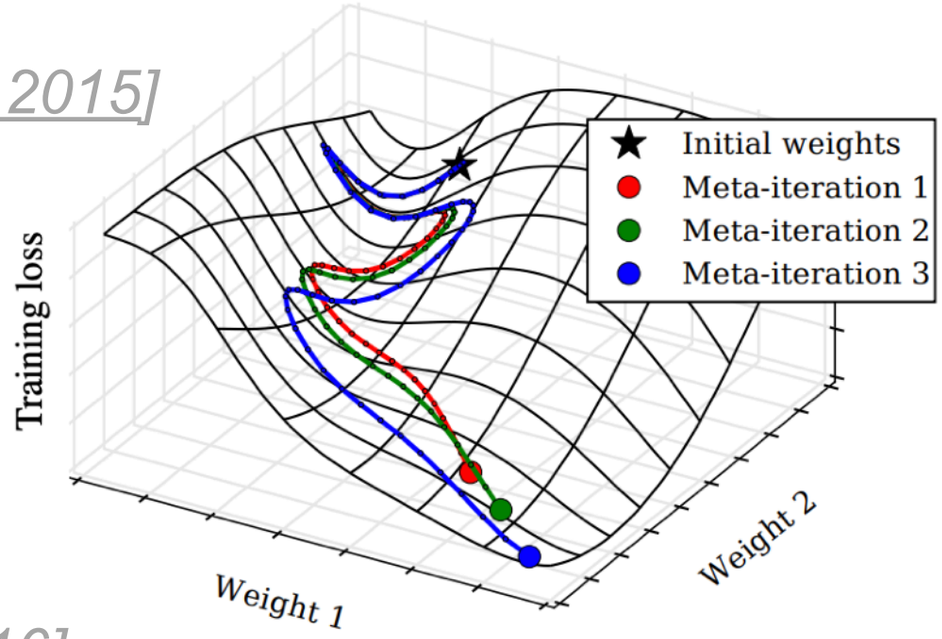
Hyperparameter gradient descent

Optimize neural network hyperparameters and weights simultaneously

- Bilevel program *[Franceschi et al 2018]*
  - Outer obj.: optimize λ
  - Inner obj.: optimize weights given λ

$$\begin{aligned} &\min_{\lambda} \ \mathcal{L}_{val}\left(w^*(\lambda), \lambda\right) \\ s.t. \quad &w^*(\lambda) = \operatorname{argmin}_w \mathcal{L}_{train}\left(w, \lambda\right) \end{aligned}$$

- Derive through the entire SGD *[MacLaurin et al 2015]*
  - Get hypergradients wrt. validation loss
  - Expensive! But useful if you have many hyper parameters and high parallelism



- Interleave optimization steps *[Luketina et al 2016]*
  - Alternate SGD steps for ω and λ

Hyperparameter gradient step w.r.t. $\nabla_{\lambda}\mathcal{L}_{val}$

Parameter gradient step w.r.t. $\nabla_{w}\mathcal{L}_{train}$

Meta-learning

Meta-learning can drastically speed up the architecture and hyperparameter search, in combination with the optimization techniques we just discussed

- Learn which hyperparameters are really important
- Learn which hyperparameter values should be tried first
- Learn which architectures will most likely work
- Learn how to clean and annotate data
- Learn which feature representations to use
- …

Thank you! To be continued…

Part 2: meta-learning

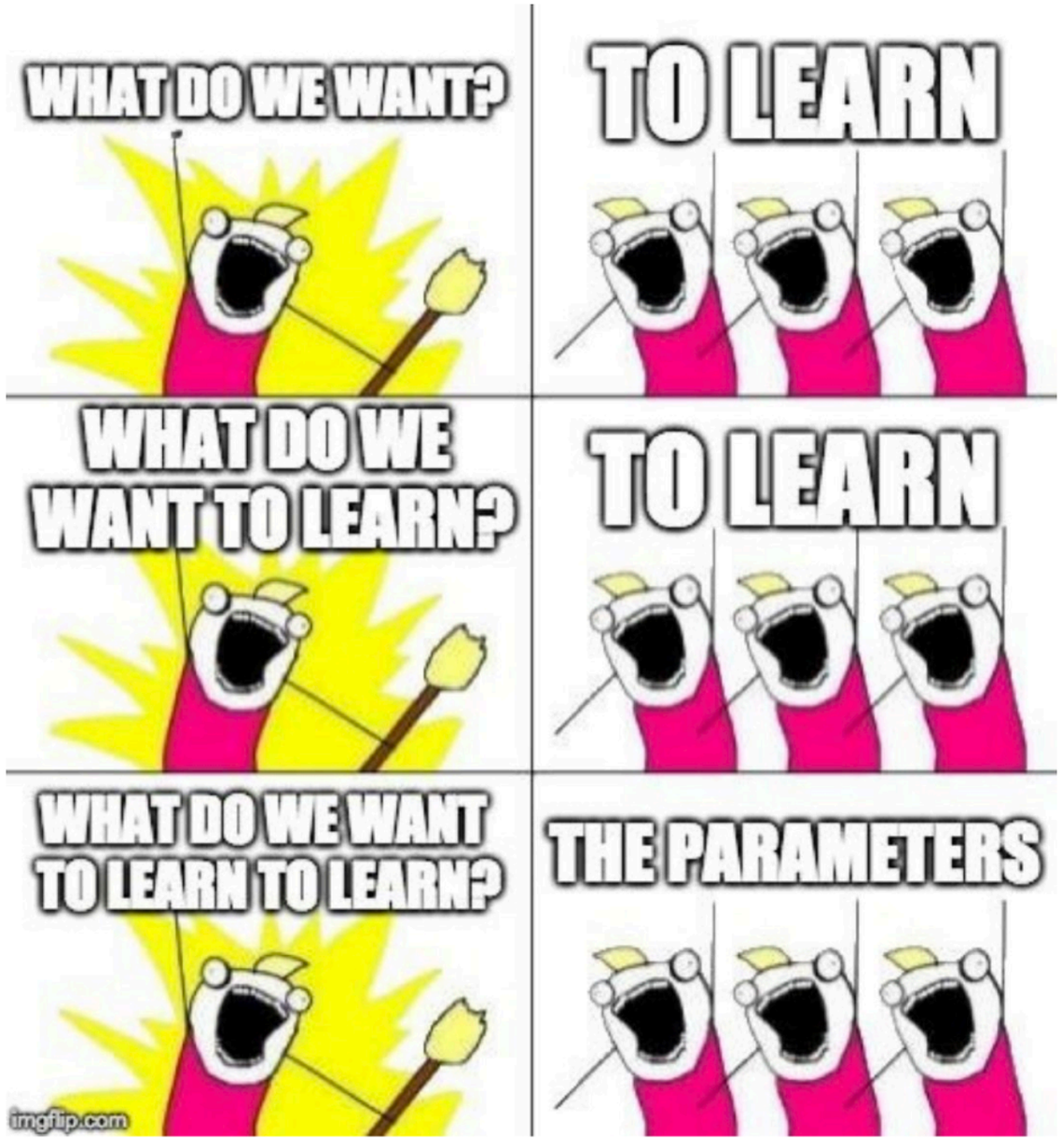
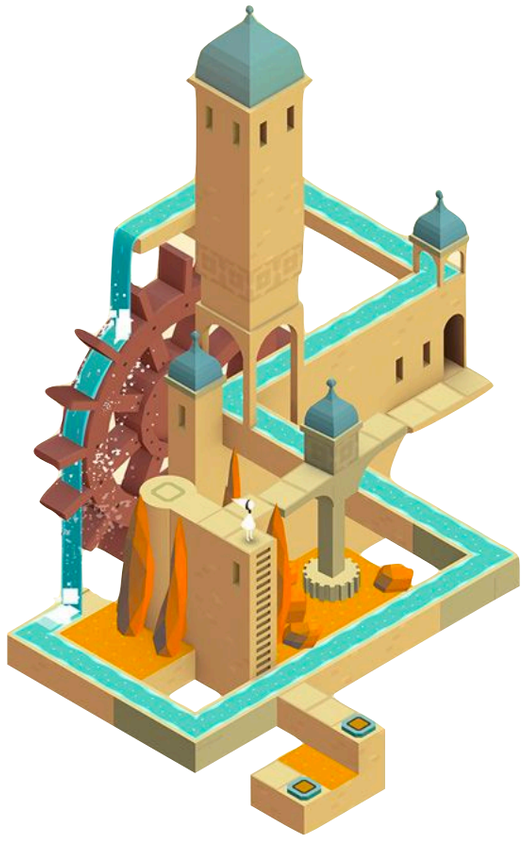


Image: Pesah et al. 2018

Advanced Course on Data Science & Machine Learning
Siena, 2019



Automatic Machine Learning & Meta-Learning

part 2

Joaquin Vanschoren

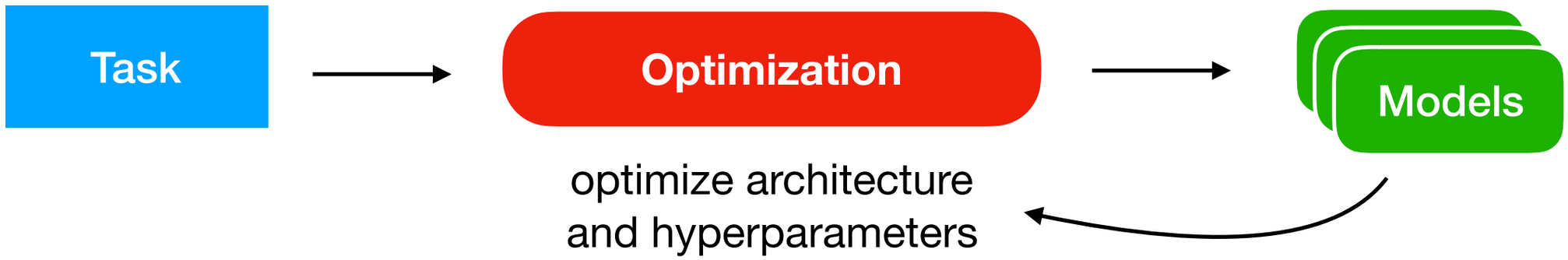
Eindhoven University of Technology

j.vanschoren@tue.nl @joavanschoren http://bit.ly/openml

slides!

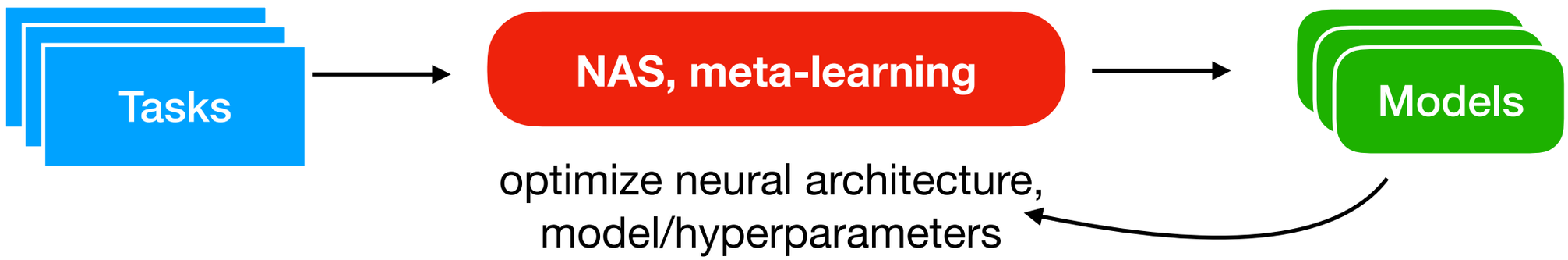
Overview

part 1 AutoML introduction, promising optimization techniques



part 2 Meta-learning

part 3 Neural architecture search, meta-learning on neural nets



Learning is a never-ending process

Humans don’t learn from scratch

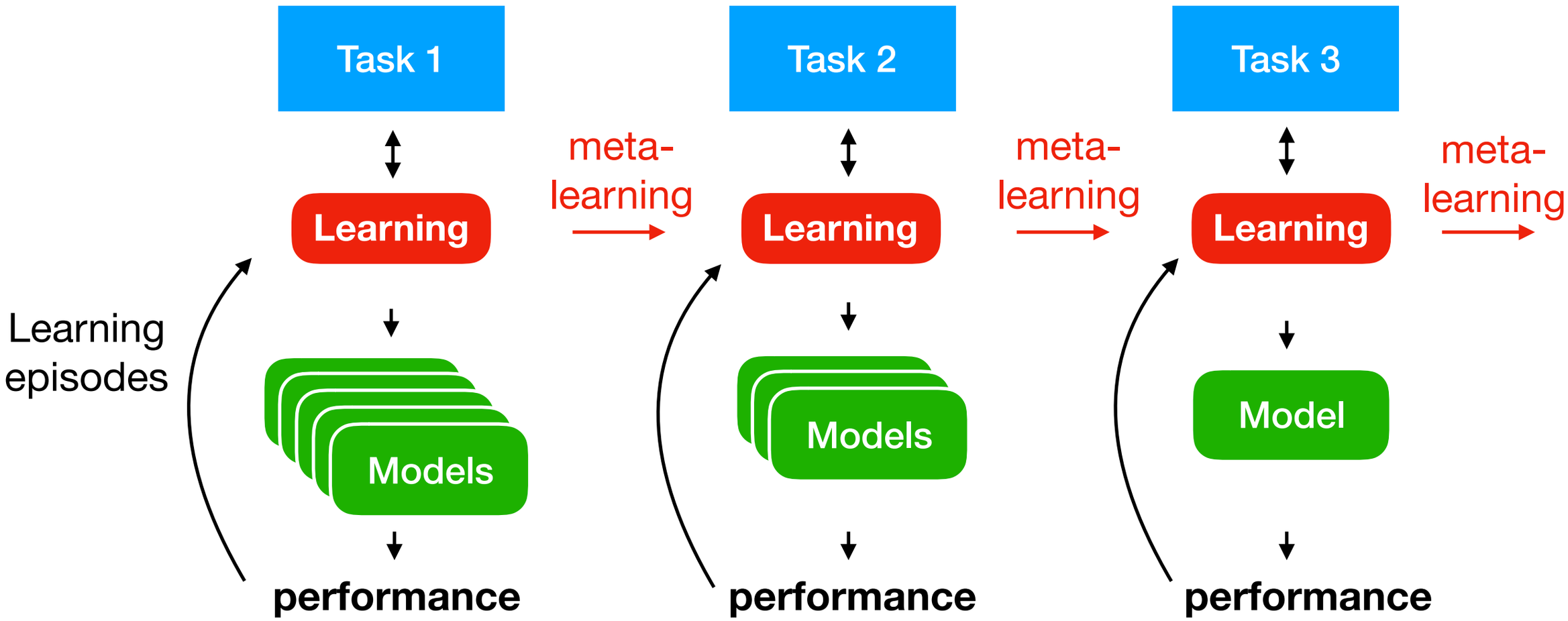
Learning is a never-ending process

Learning humans also seek/create related tasks

E.g. find many similar puzzles, solve them in different ways,…

Learning is a never-ending process

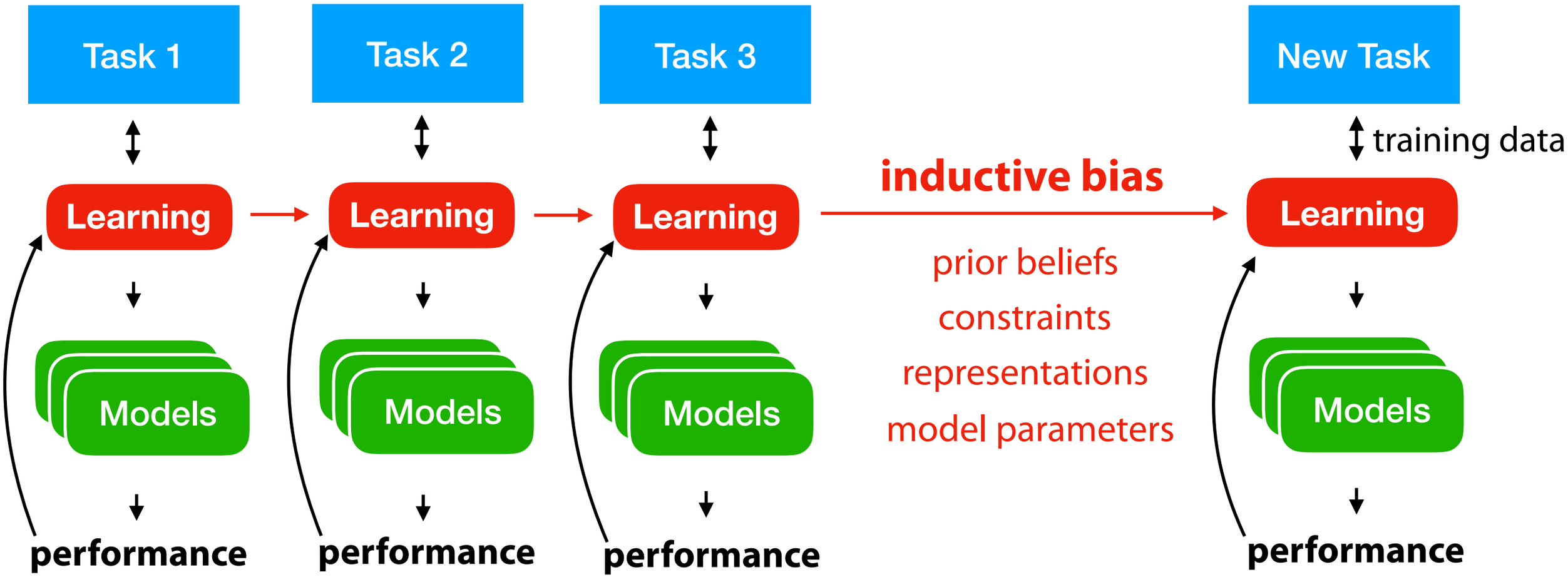
Humans learn *across* tasks
Why? Requires less trial-and-error, less data



Learning to learn

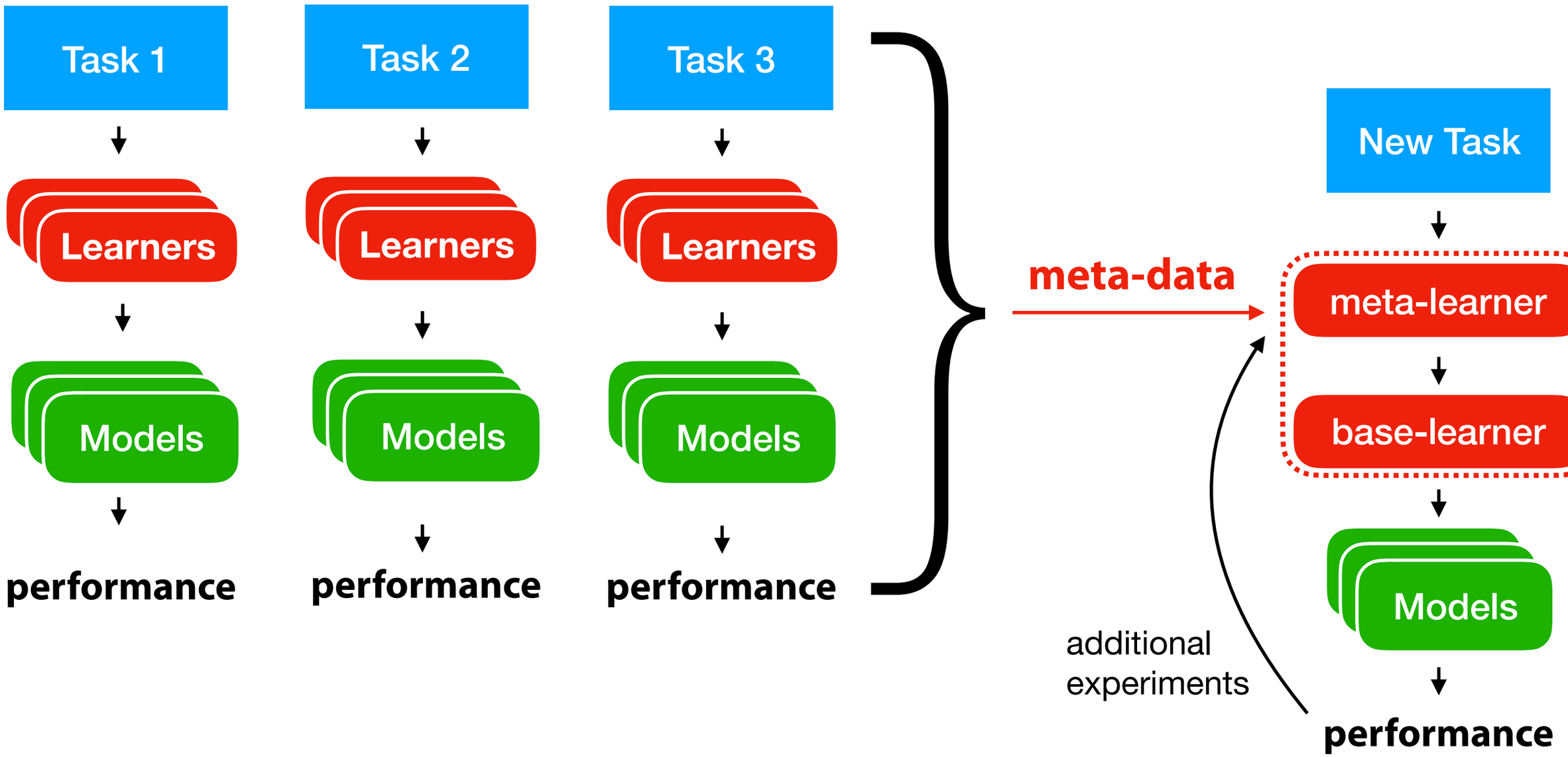
If prior tasks are *similar*, we can ***transfer*** prior knowledge to new tasks

Inductive bias: assumptions added to the training data to learn effectively



Meta-learning

Meta-learner *learns* a (base-)learning algorithm, based on *meta-data*



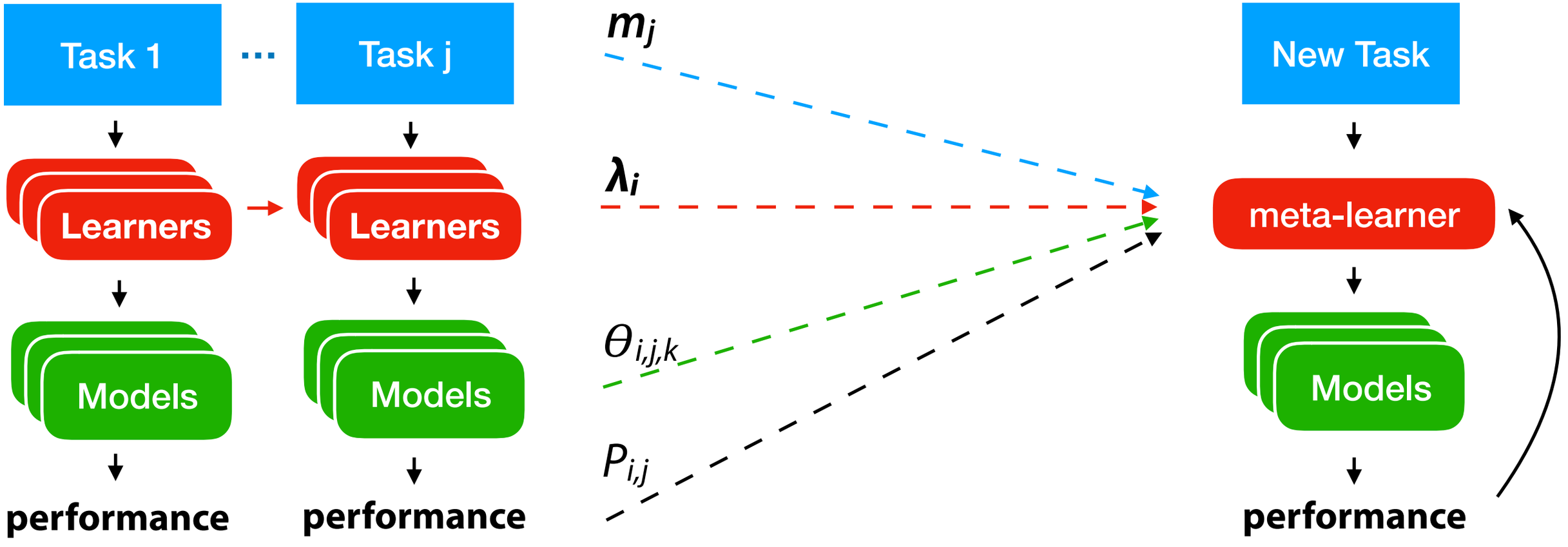
Meta-data

$\boldsymbol{m_j}$ Description of task j (meta-features)

$\boldsymbol{\lambda_i}$ Configuration i (architecture + hyperparameters)

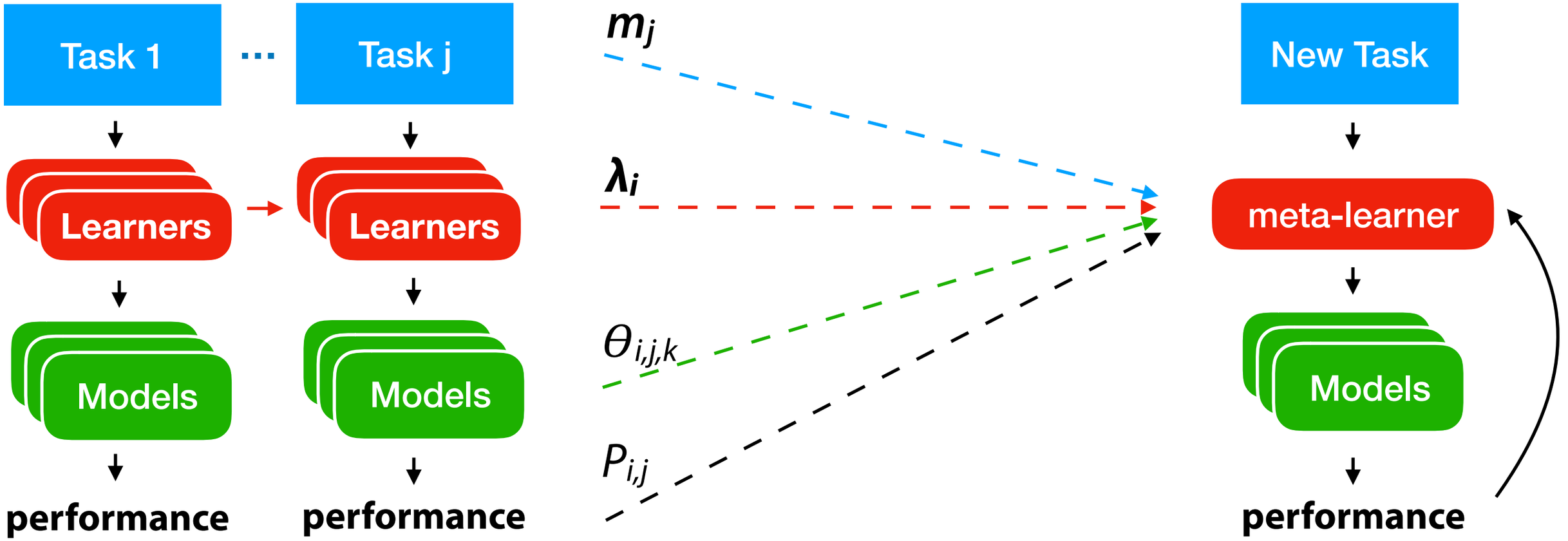
$\theta_{i,j,k}$ Model parameters (e.g. weights)

$P_{i,j}$ Performance estimate of model built with λ_i *on task j*



How?

1. Learn from observing learning algorithms (for *any* task) $\boldsymbol{\lambda_i}$ $P_{i,j}$
2. Learn what may likely work (for *somewhat similar* tasks) $\boldsymbol{m_j}$ $\boldsymbol{\lambda_i}$ $P_{i,j}$
3. Learn from previous models (for *very similar* tasks) $\theta_{i,j,k}$ $\boldsymbol{(m_j)}$ $\boldsymbol{\lambda_i}$ $P_{i,j}$

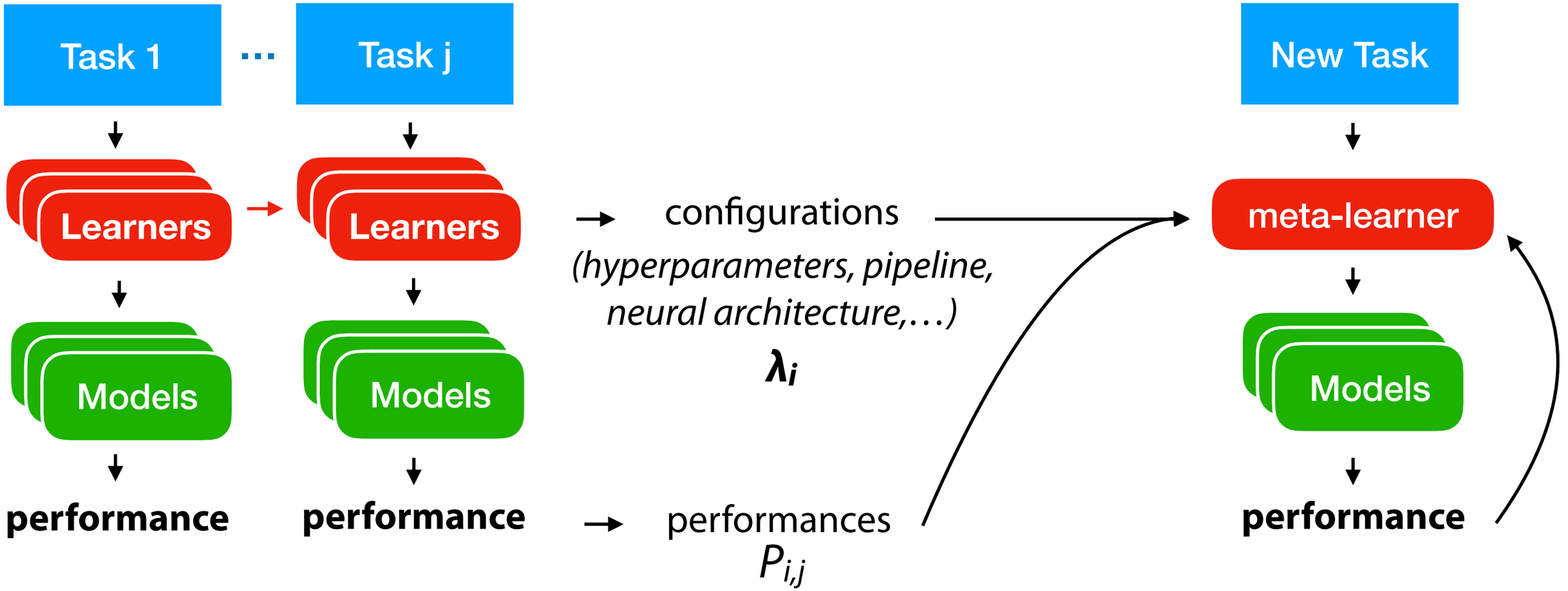


1. Learn from observing learning algorithms

Define, store and use meta-data:

- ***configurations***: settings that uniquely define the model

- ***performance*** (e.g. accuracy) on specific tasks

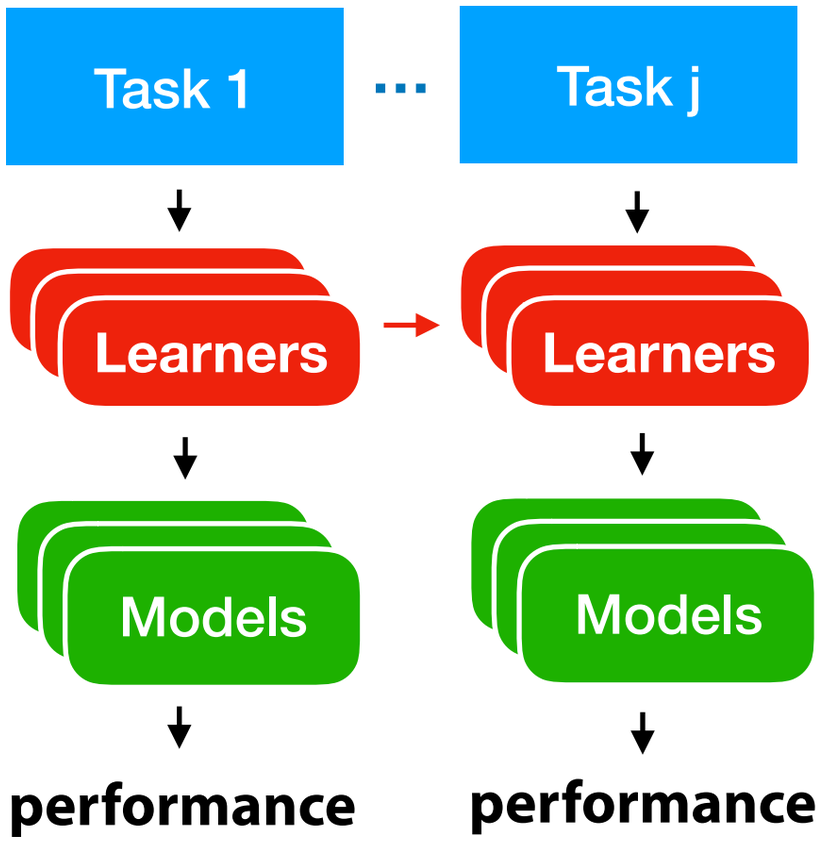


1. Learn from observing learning algorithms

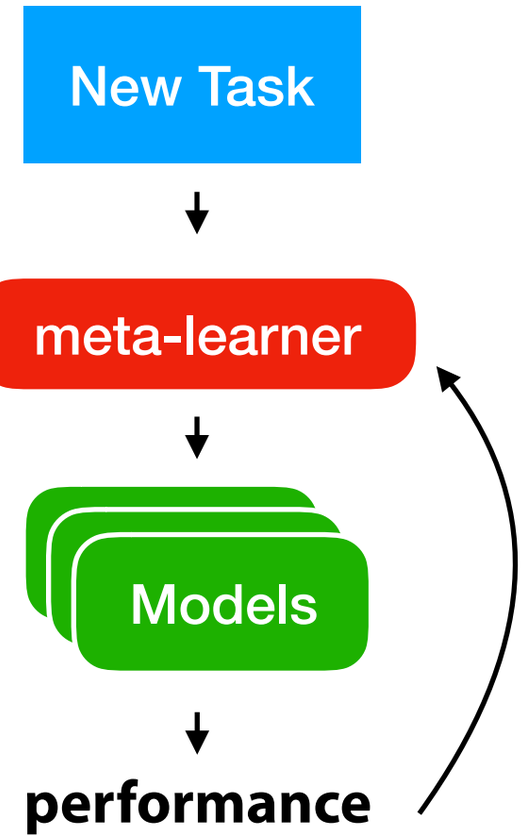
Define, store and use meta-data:

- ***configurations***: settings that uniquely define the model

- ***performance*** (e.g. accuracy) on specific tasks

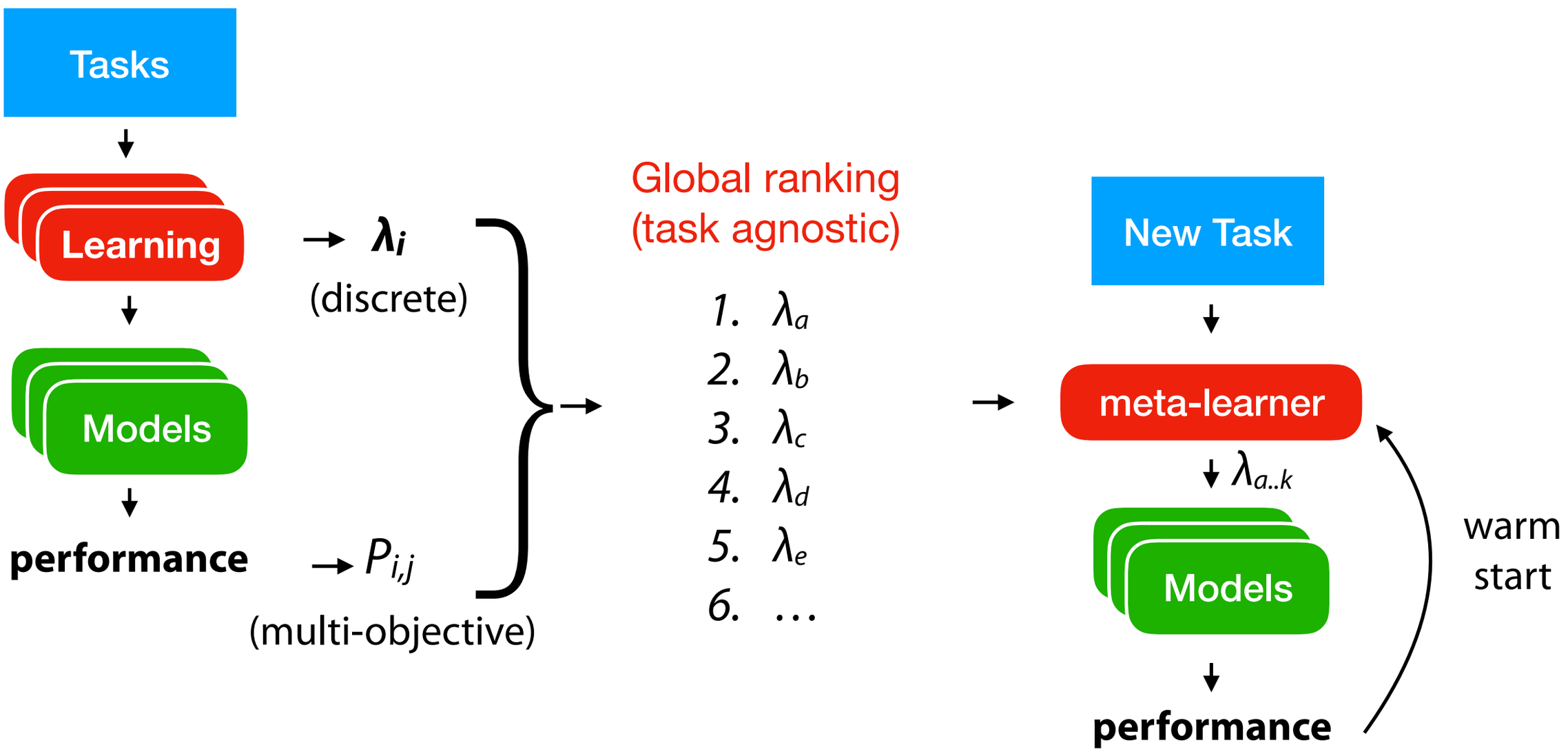


task	model config λ	score P
0	alg=SVM, C=0.1	0,876
0	alg=NN, lr=0.9	0,921
0	alg=RF, mtry=0.1	0,936
1	alg=SVM, C=0.2	0,674
1	alg=NN, lr=0.5	0,777
1	alg=RF, mtry=0.4	0,791



Rankings

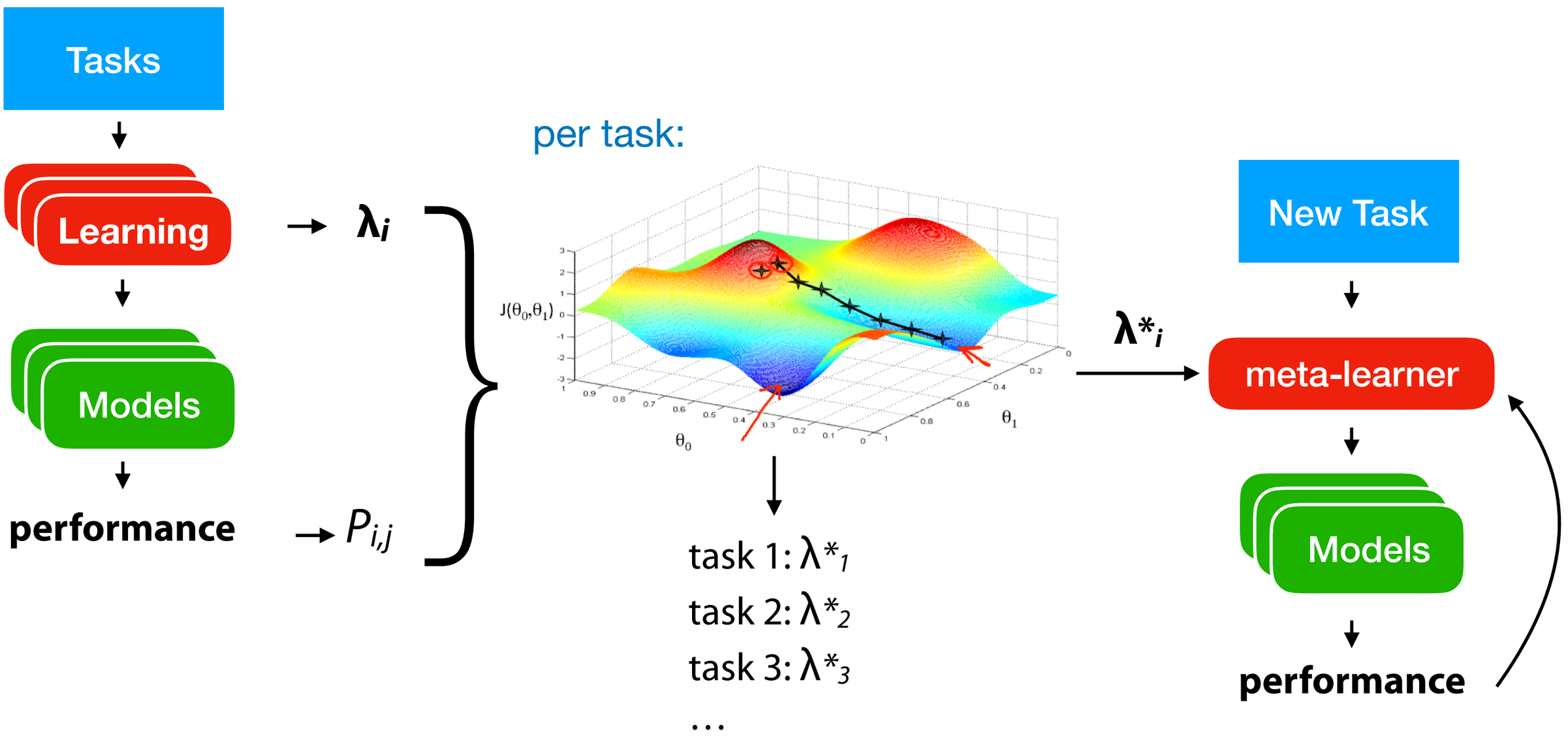
- Build a *global (multi-objective) ranking,* recommend the top-K
- Can be used as a *warm start* for optimization techniques
  - E.g. Bayesian optimization, evolutionary techniques,…



Wistuba et al. 2015

Warm-starting with plugin estimators

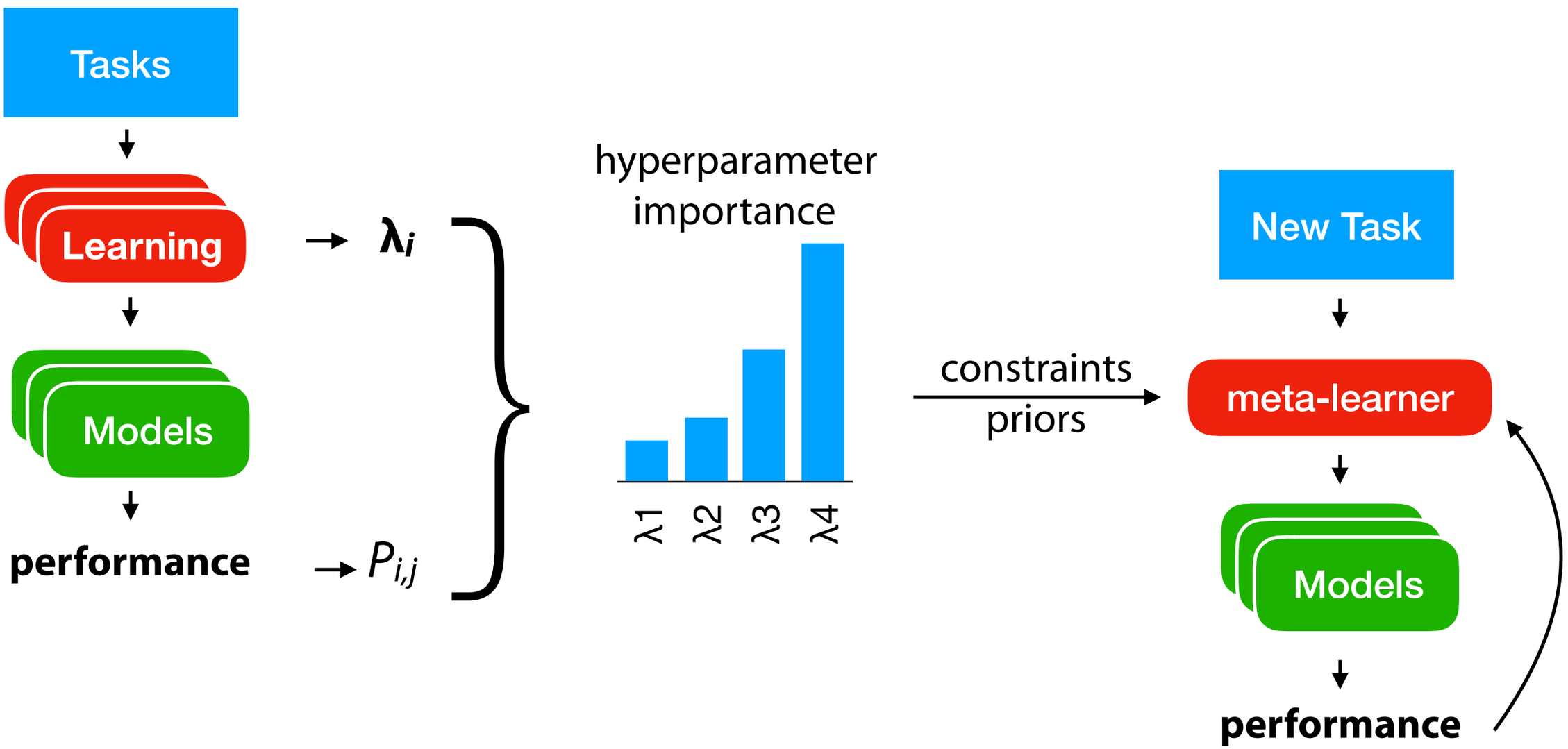
- What if prior configurations are not optimal?
- Per task, fit a differentiable plugin estimator on all evaluated configurations
- Do gradient descent to find optimized configurations, recommend those



[1] van Rijn & Hutter 2018

Hyperparameter behavior

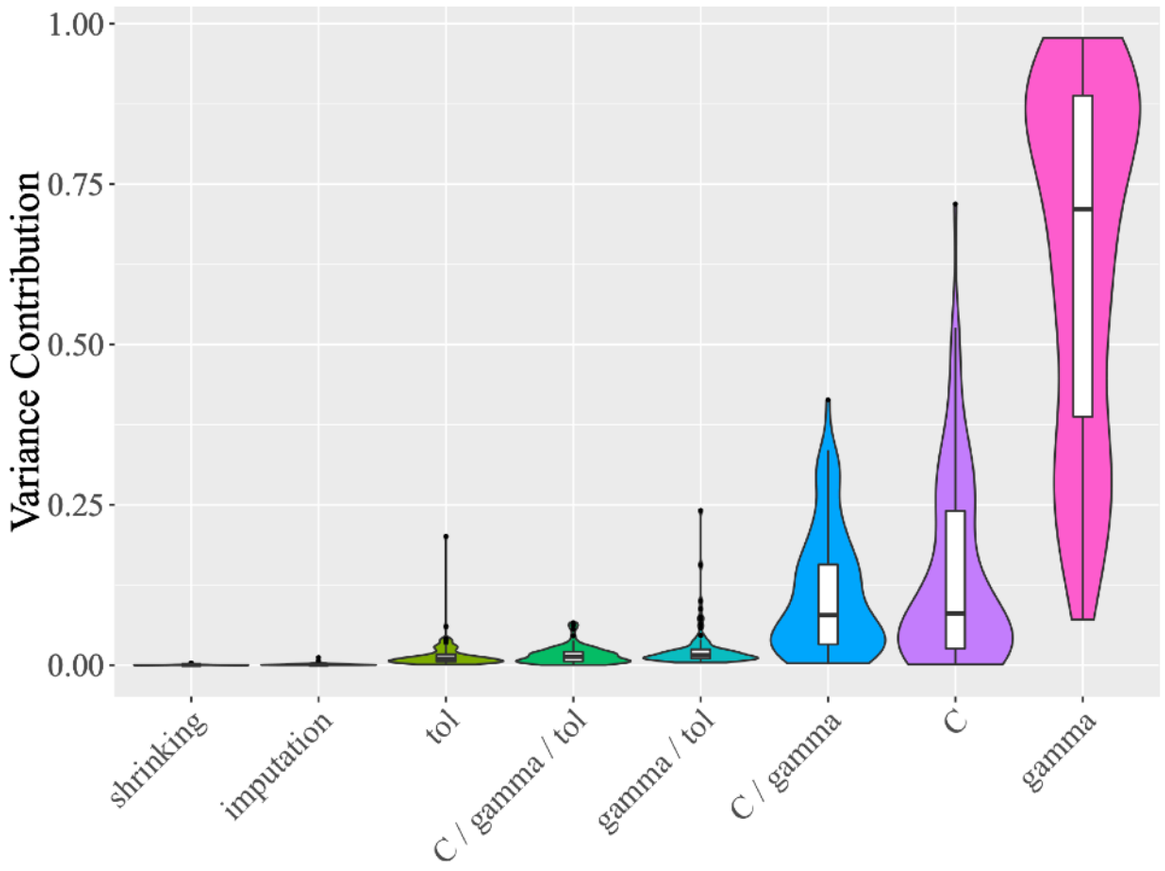
- **Functional ANOVA** [1]
  Select hyperparameters that cause variance in the evaluations.



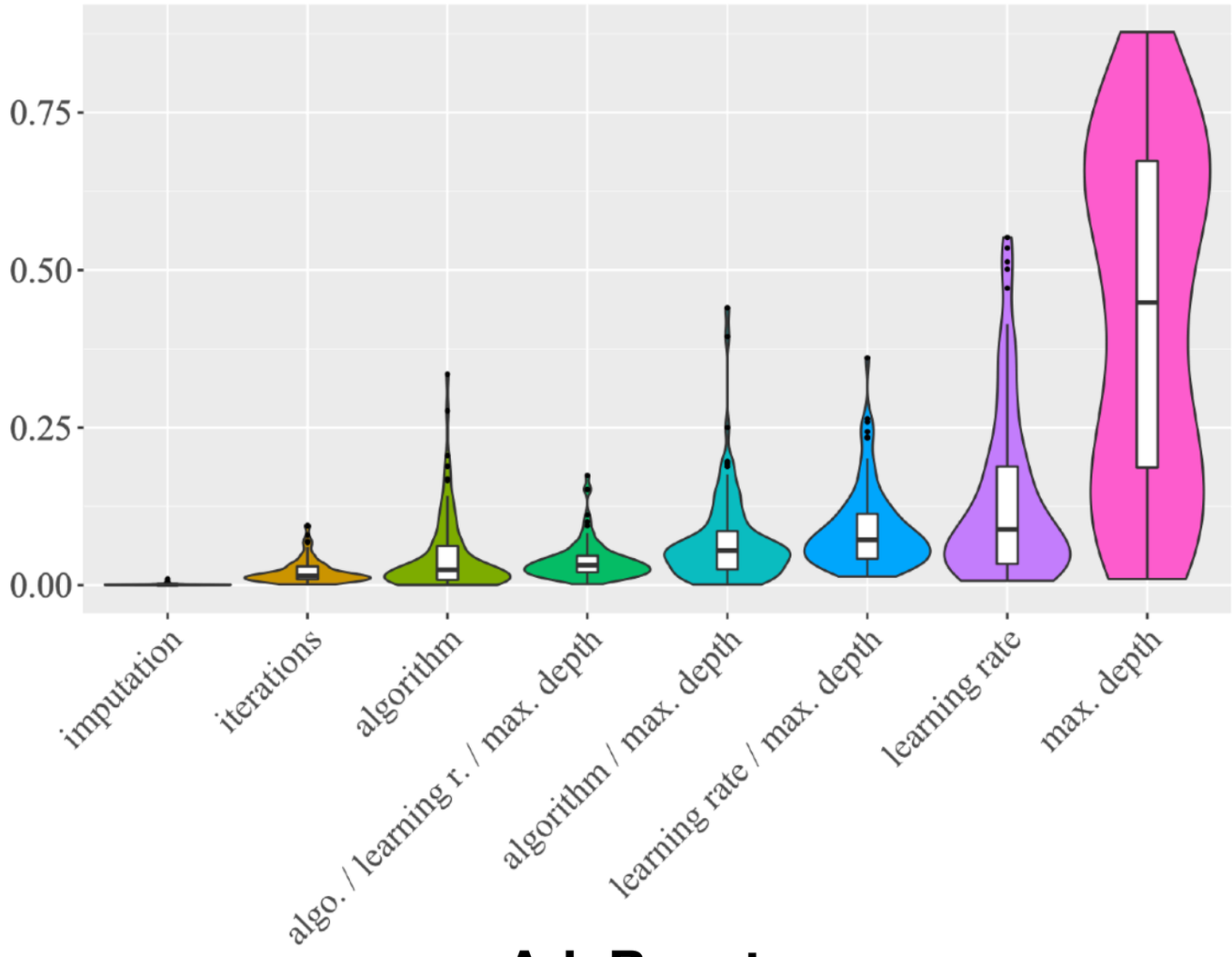
[1] van Rijn & Hutter 2018

Hyperparameter behavior

- **Functional ANOVA** [1]
  Select hyperparameters that cause variance in the evaluations.



SVM (RBF)

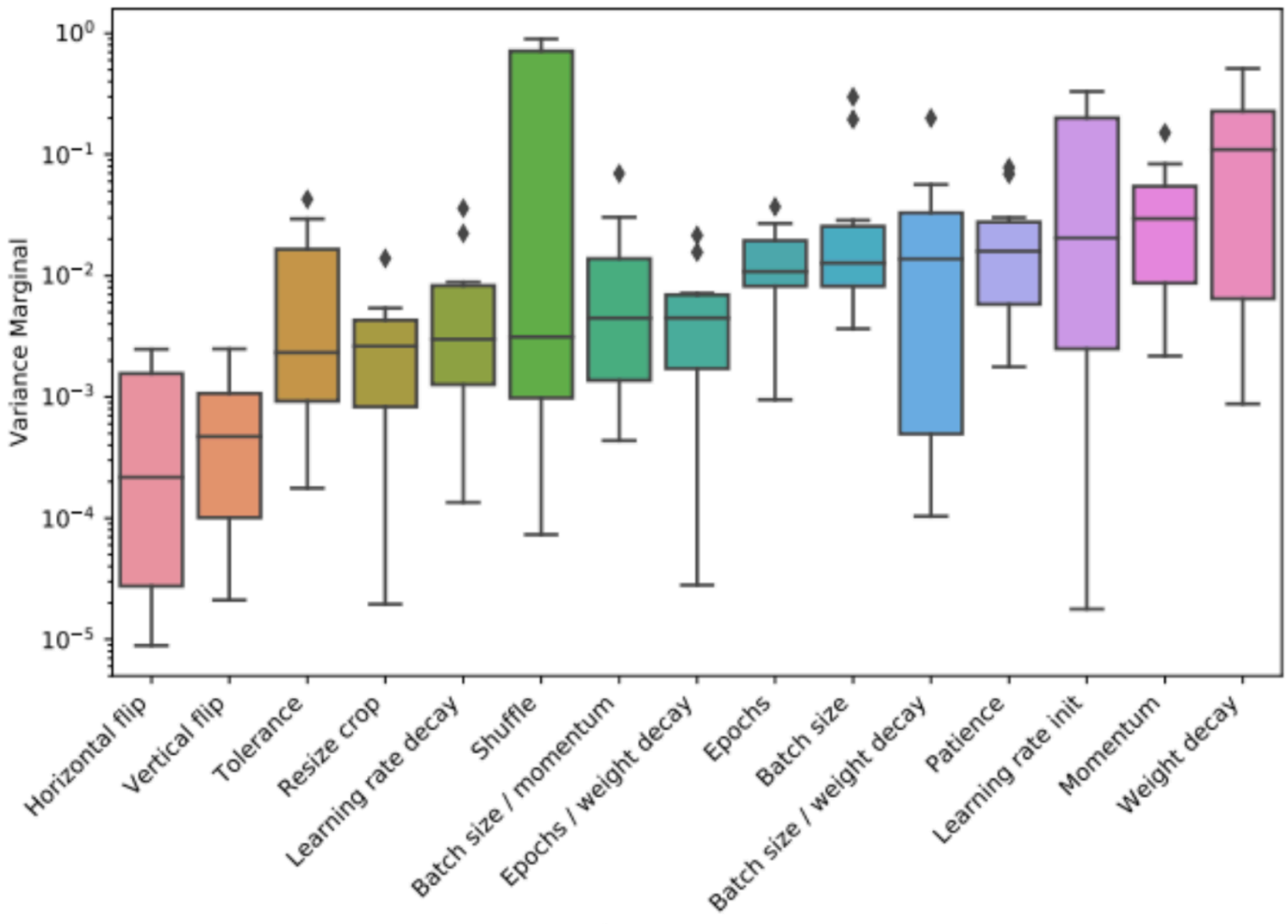


AdaBoost

[1] van Rijn & Hutter 2018

Hyperparameter behavior

- **Functional ANOVA** [1]
  Select hyperparameters that cause variance in the evaluations.

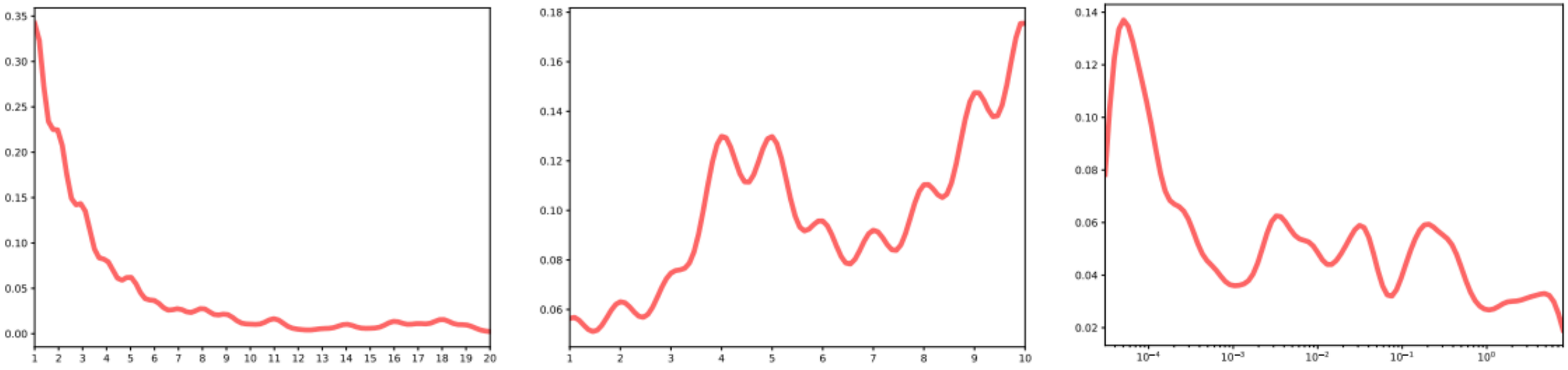


ResNets for image classification *(under review)*

[1] van Rijn & Hutter 2018

Hyperparameter behavior

- **Functional ANOVA** [1]
  Select hyperparameters that cause variance in the evaluations.
  Learn priors for hyperparameter tuning (e.g. Bayesian optimization)



(a) RF: min. samples per leaf

(b) Adaboost: max. depth of tree

(c) SVM (RBF kernel): gamma

Priors (KDE) for the most important hyperparameters

[1] van Rijn & Hutter 2018

[2] Probst et al. 2018

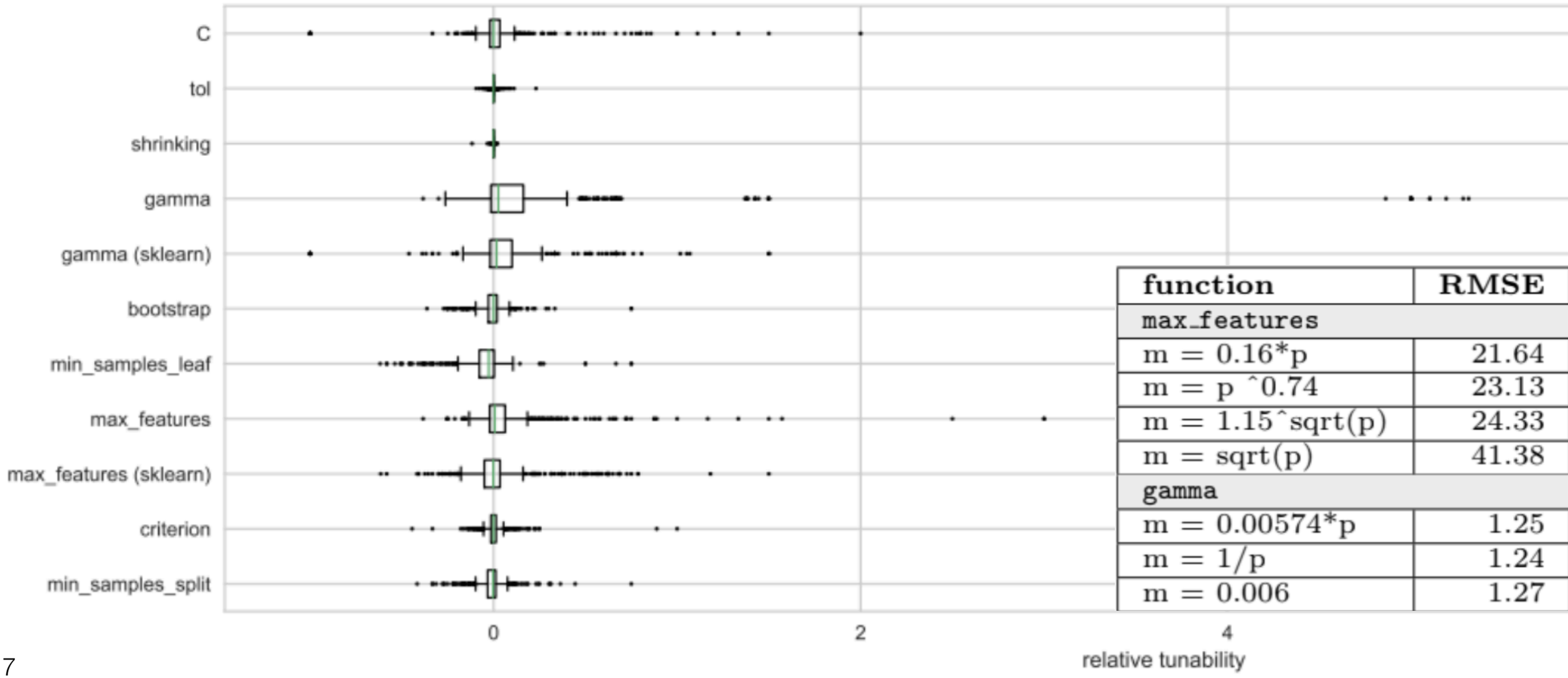
[3] Weerts et al. 2018

[4] van Rijn et al. 2018

Hyperparameter behavior

- **Functional ANOVA** [1]
  Select hyperparameters that cause variance in the evaluations.

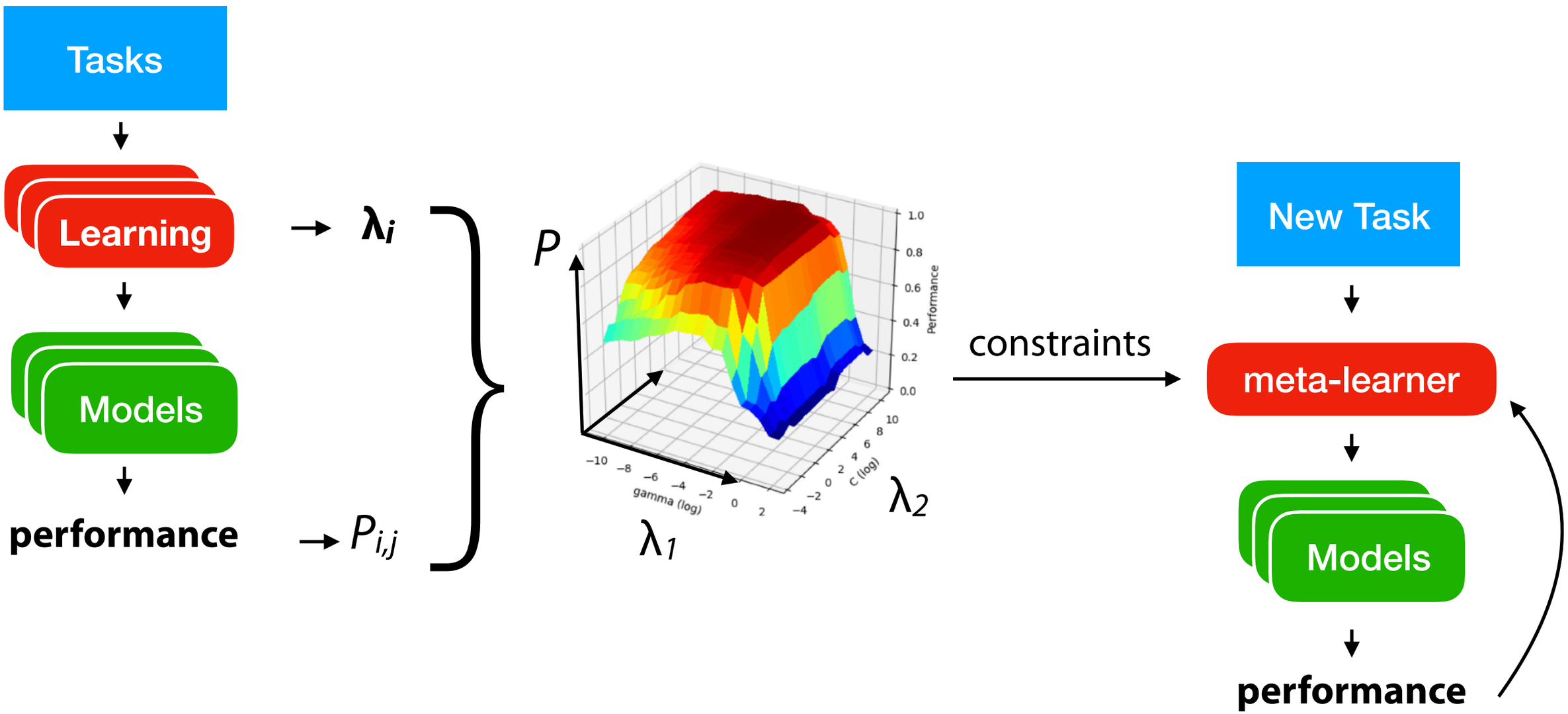
- **Tunability** [2,3,4]
  Learn good defaults, measure improvement from tuning over defaults



function	RMSE
max_features	
m = 0.16*p	21.64
m = p ^0.74	23.13
m = 1.15^sqrt(p)	24.33
m = sqrt(p)	41.38
gamma	
m = 0.00574*p	1.25
m = 1/p	1.24
m = 0.006	1.27

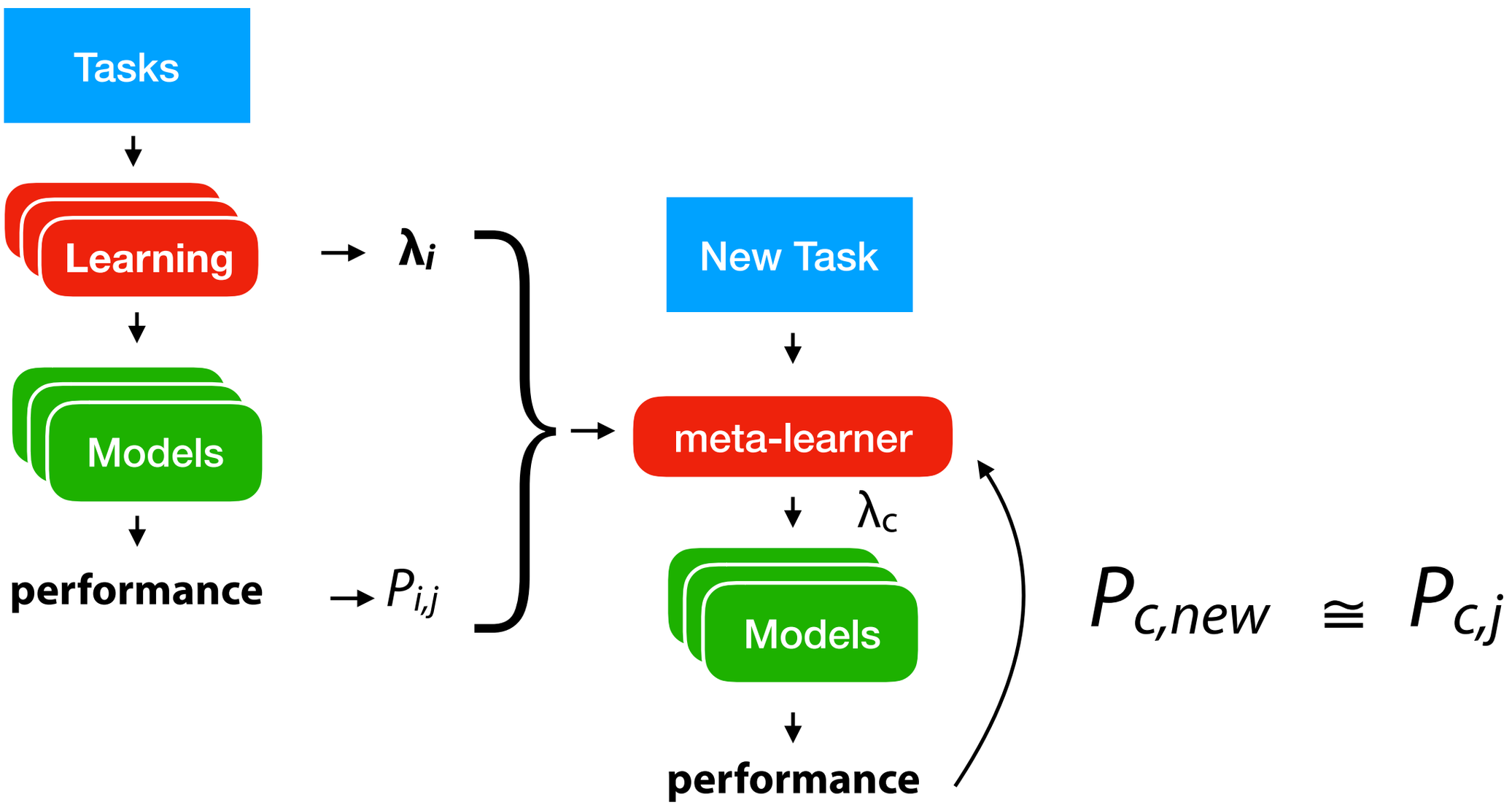
Hyperparameter behavior

- **Search space pruning**
  Exclude regions yielding bad performance on (similar) tasks



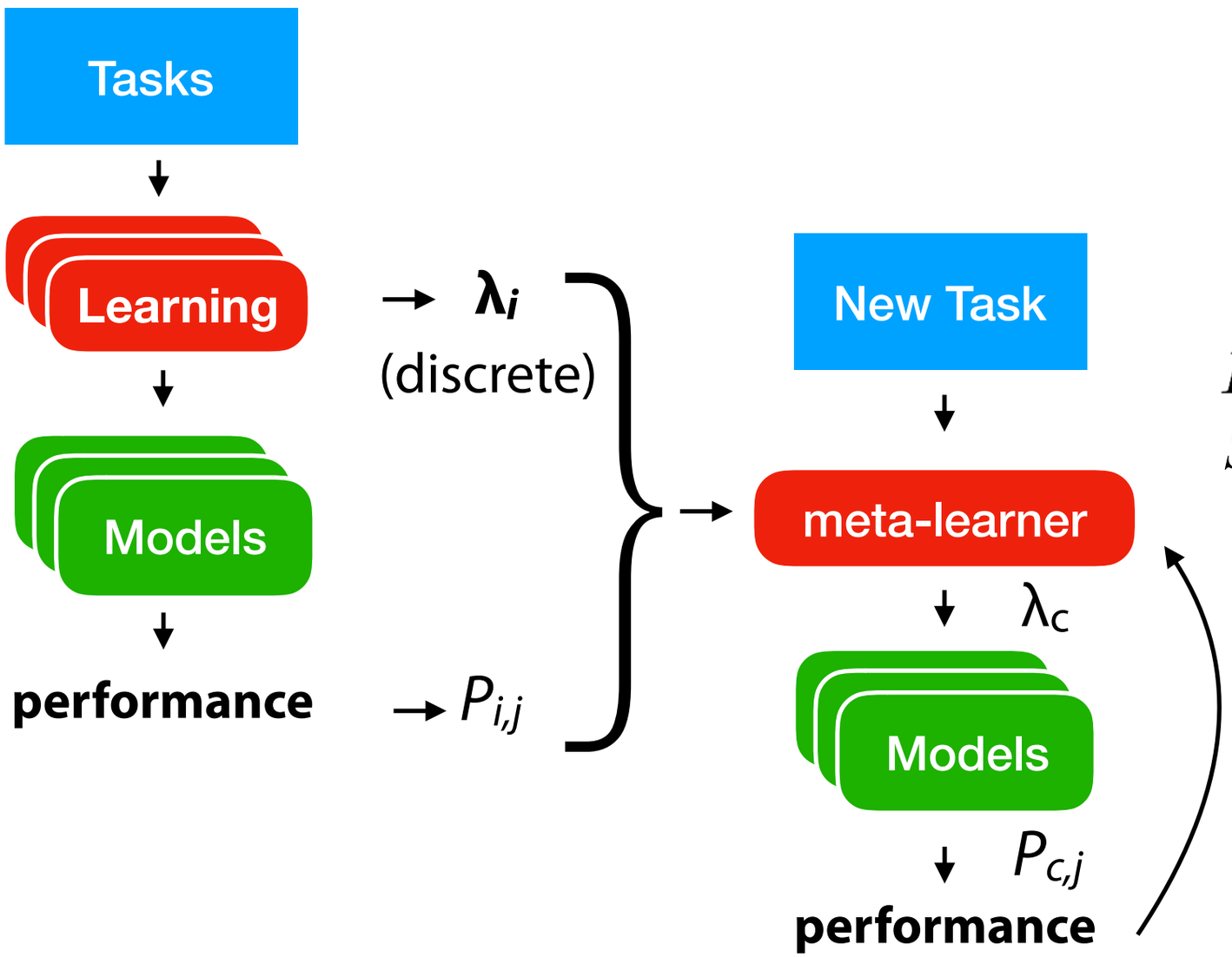
Learning task similarity

- Experiments on the *new* task can tell us how it is similar to *previous* tasks
- **Task are *similar*** if observed *performance* of configurations is similar
- Use this to recommend new configurations, run, repeat



Active testing

- Learn task similarity while tuning configurations
- Tournament-style selection, warm-start with overall best config λ_{best}
- Next candidate λ_c : the one that beats current λ_{best} on similar tasks



$$RL_{a,b,j} = P_{a,j} - P_{b,j}$$
$$Sim(t_j, t_{new}) = Corr([RL_{a,b,j}], [RL_{a,b,new}])$$

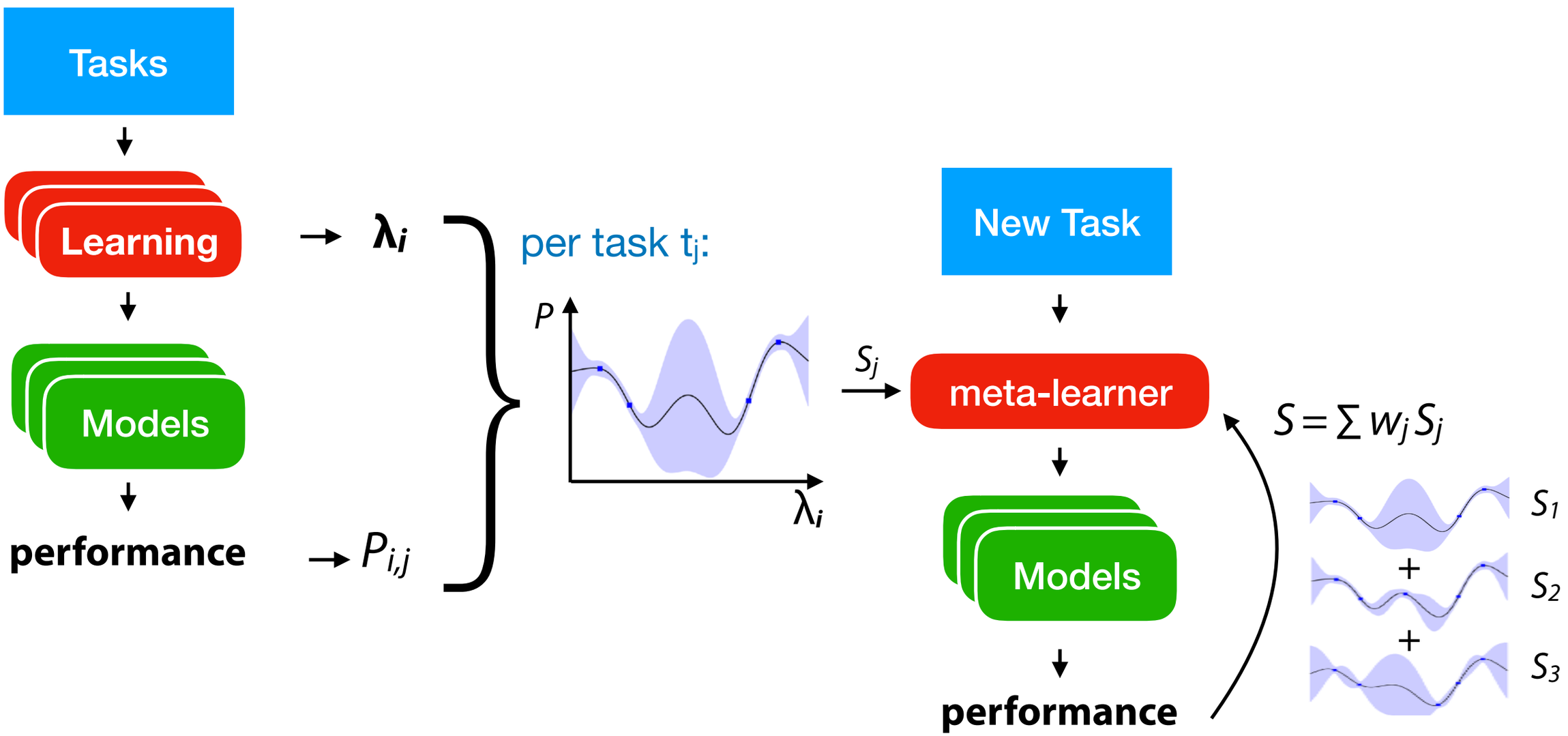
Select $\lambda_c > \lambda_{best}$ on similar tasks

[1] Wistuba et al. 2018

[2] Feurer et al. 2018

Surrogate model transfer

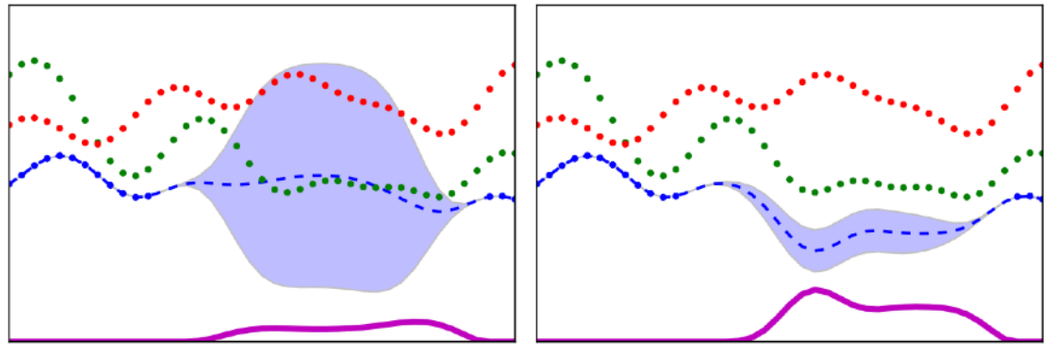
- If task j is *similar* to the new task, its surrogate model S_j will likely transfer well
- Sum up all S_j predictions, weighted by task similarity (as in active testing)[1]
- Build combined Gaussian process, *weighted by current performance* on new task[2]



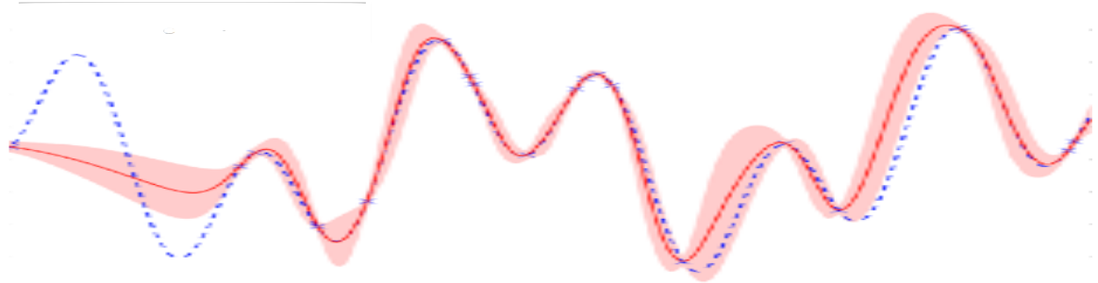
[1] Swersky et al. 2013
[2] Springenberg et al. 2016
[3] Golovin et al. 2017

Multi-task Bayesian optimization

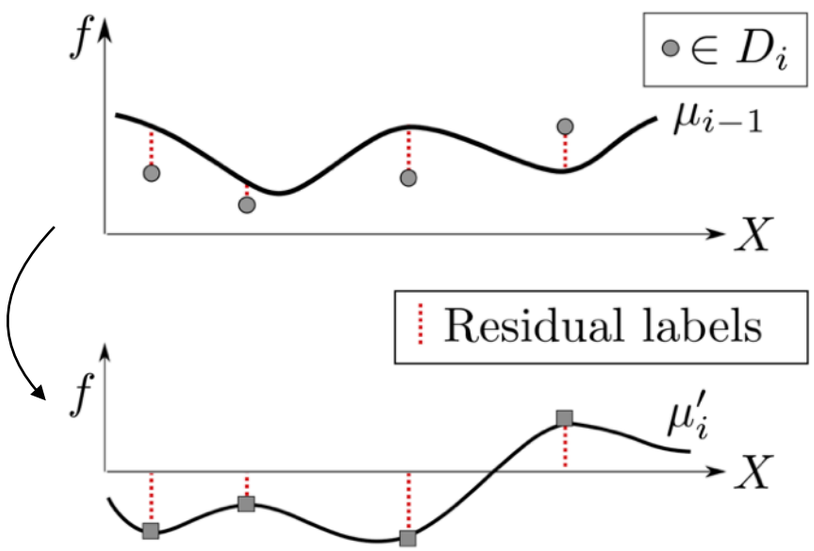
- **Multi-task Gaussian processes:** train surrogate model on t tasks simultaneously[1]
  - If tasks are similar: transfers useful info
  - Not very scalable



- **Bayesian Neural Networks** as surrogate model[2]
  - Multi-task, more scalable

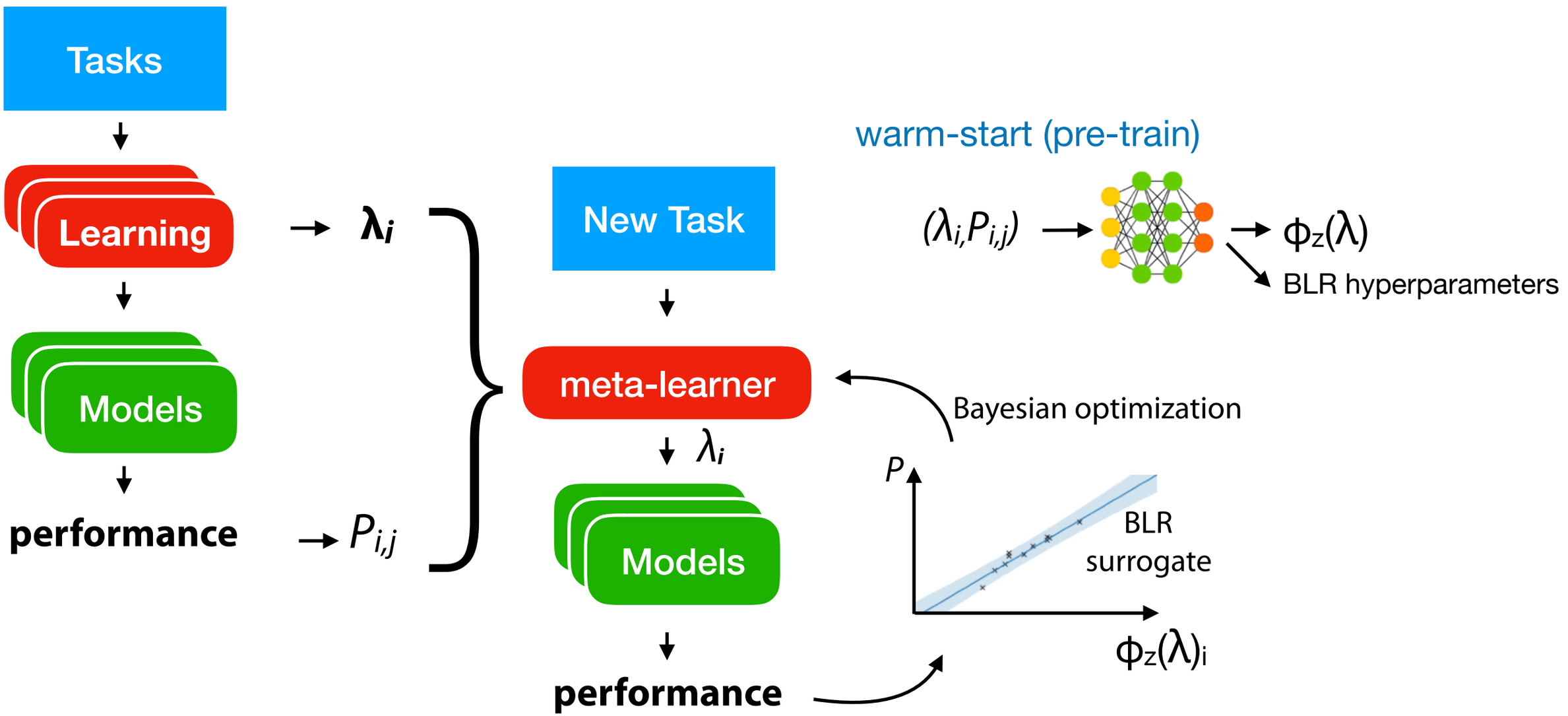


- **Stacking** Gaussian Process regressors (Google Vizier)[3]
  - Continual learning (sequential similar tasks)
  - Transfers a prior based on residuals of previous GP



More scalable variants

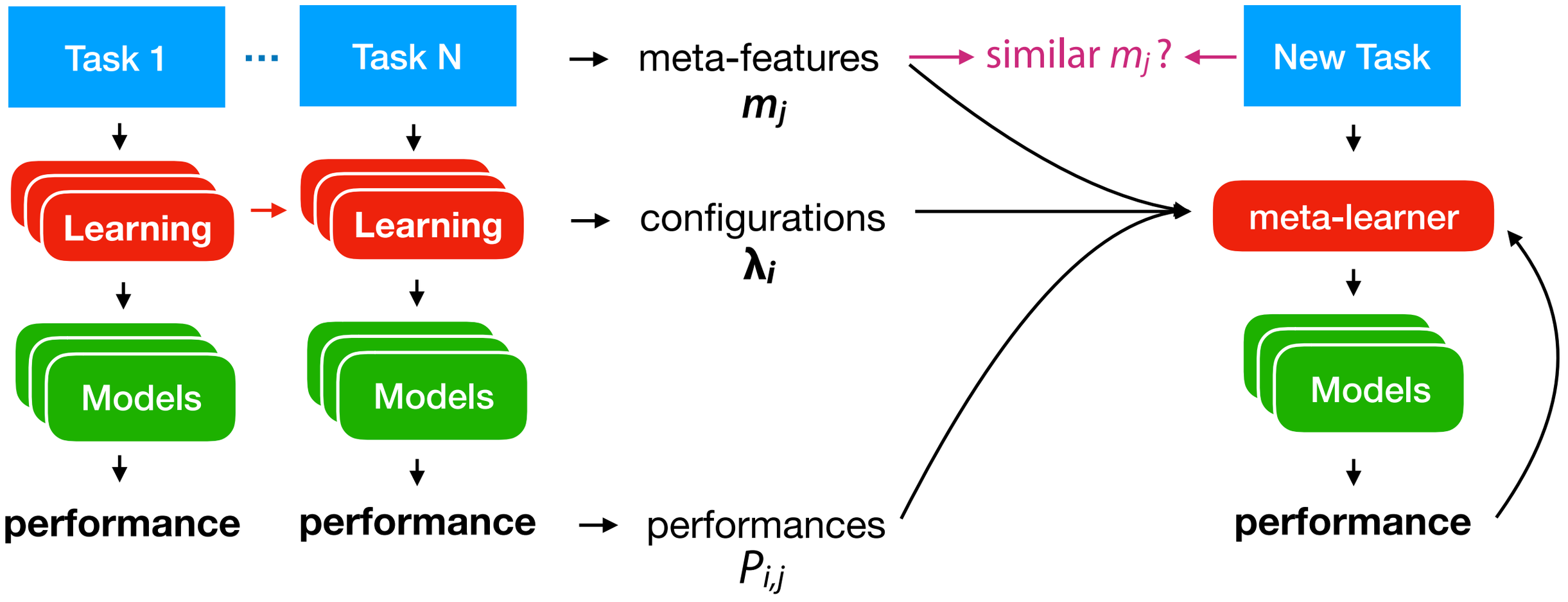
- Bayesian linear regression (BLR) surrogate model on every task
- Use neural net to learn a suitable basis expansion $\phi_z(\lambda)$ for all tasks
- Scales linearly in # observations, transfers info on configuration space



2. Learn what may likely work (for *partially similar* tasks)

Meta-features: measurable properties of the tasks

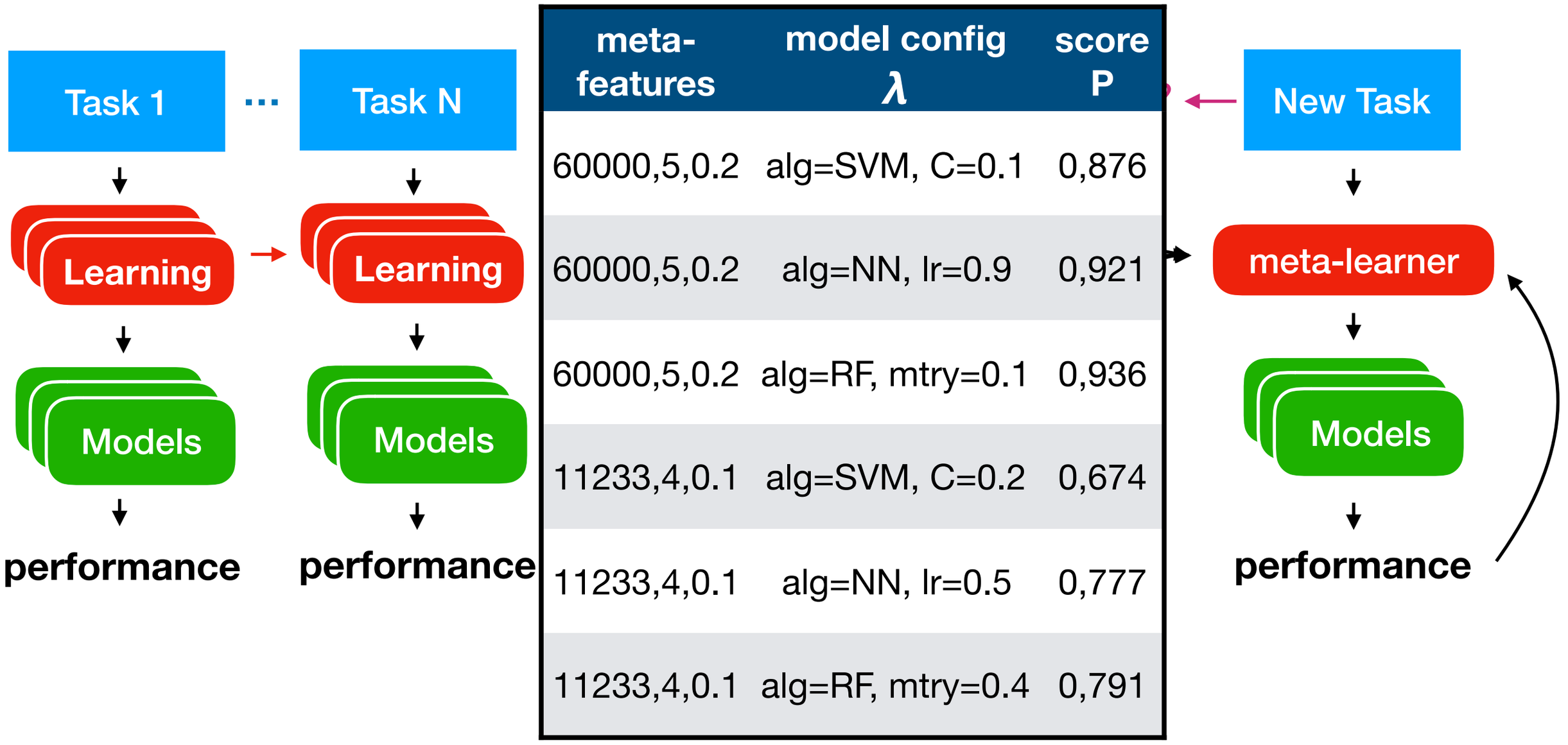
(number of instances and features, class imbalance, feature skewness,…)



2. Learn what may likely work (for *partially similar* tasks)

Meta-features: measurable properties of the tasks

(number of instances and features, class imbalance, feature skewness,…)



[1] Vanschoren 2018

[2] Rivolli et al 2019

Meta-features

- **Hand-crafted (interpretable) meta-features[1]**
  - **Number of** instances, features, classes, missing values, outliers,…
  - **Statistical:** skewness, kurtosis, correlation, covariance, sparsity, variance,…
  - **Information-theoretic**: class entropy, mutual information, noise-signal ratio,…
  - **Model-based**: properties of simple models trained on the task
  - **Landmarkers**: performance of fast algorithms trained on the task
  - Domain specific task properties
- Optimized (recommended) representations exist [2]

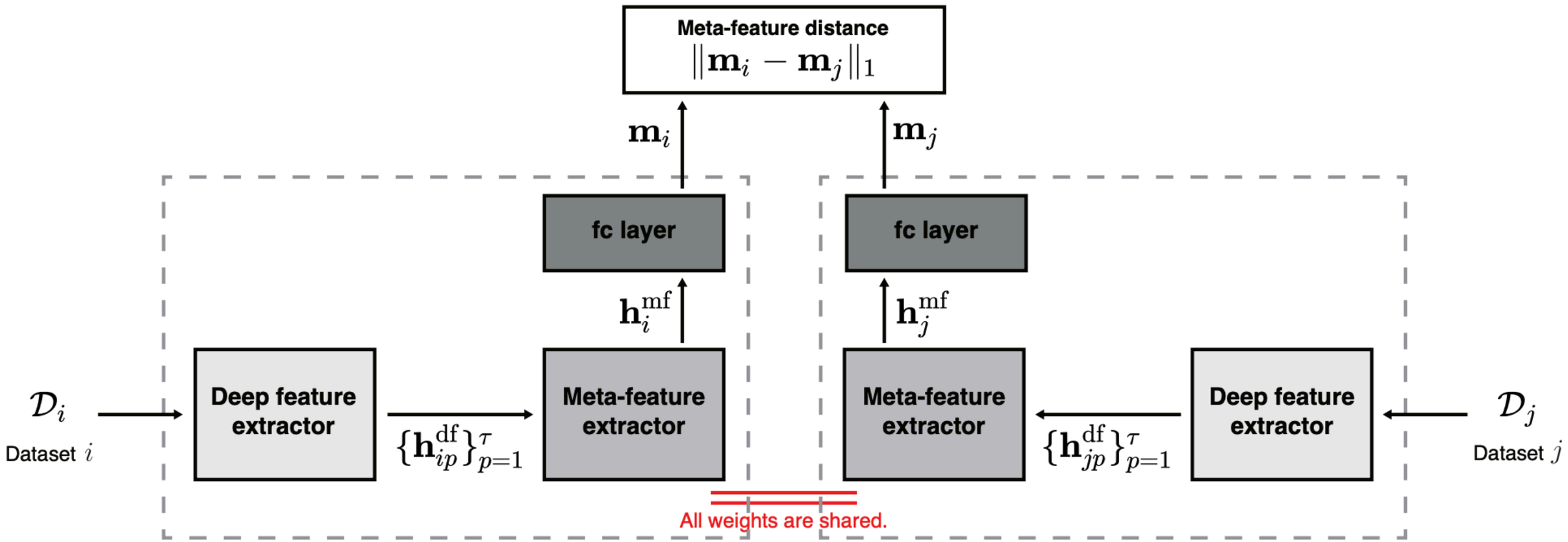
[1] Kim et al. 2017

[2] Achille et al. 2019

[2] Zamir et al. 2018

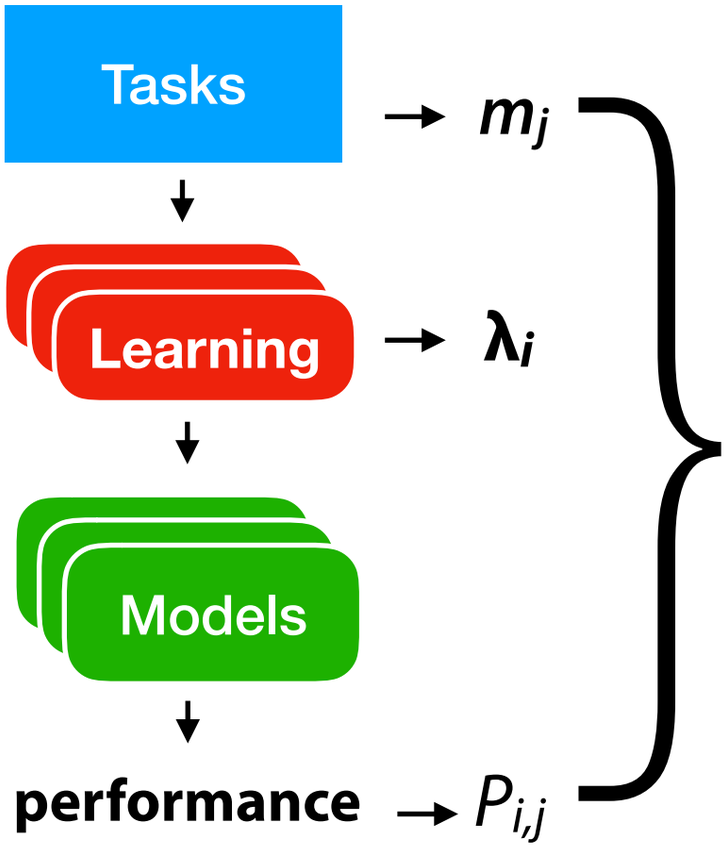
Meta-features

- ***Learning* a joint task representation**
  - Deep metric learning: learn a representation h^{mf} using ground truth distance
    - Siamese Network: similar task, similar representation [1]
    - Feed through pretrained CNN, extract vector from weights [2]
  - Recommend neural architectures, or warm-start neural architecture search
  - Ground truth can be obtained by brute force evaluation, e.g. taskonomy [3]



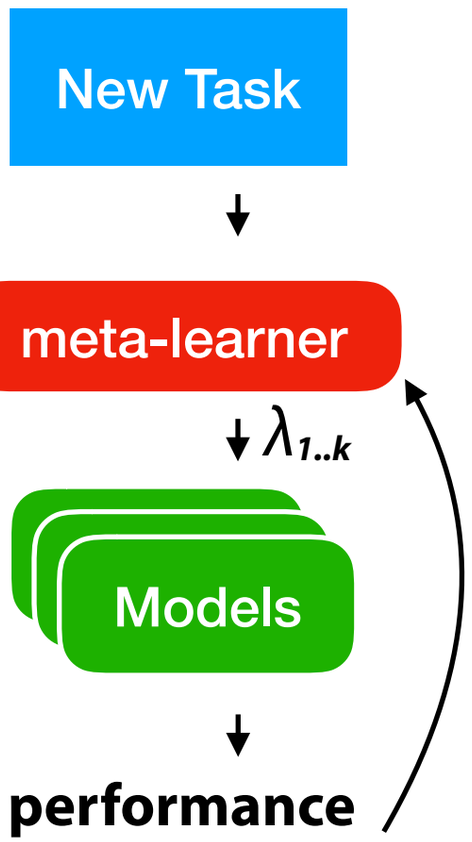
Warm-starting from similar tasks

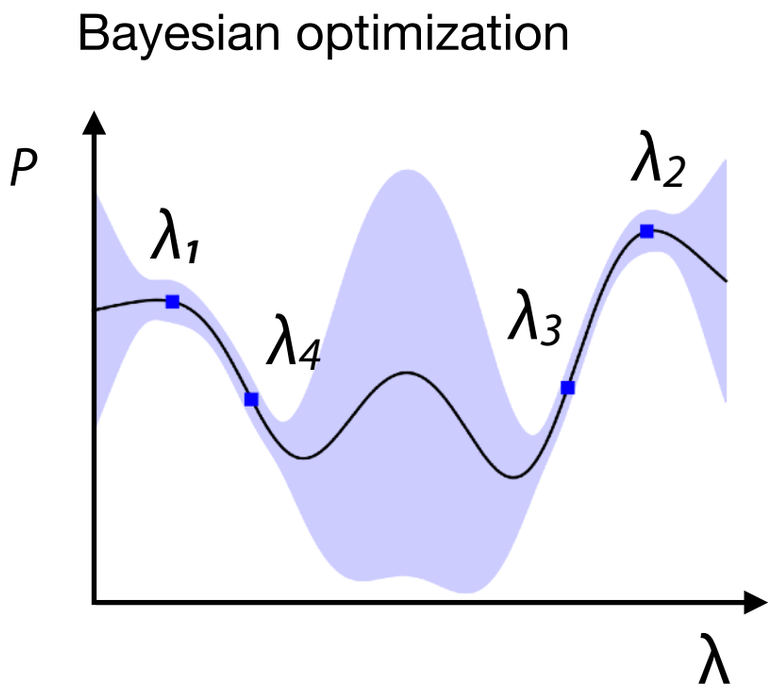
- Find k most similar tasks, warm-start search with best λ_i
  - Auto-sklearn: Bayesian optimization (SMAC)
    - Meta-learning yield better models, faster
    - Winner of AutoML Challenges



best λ_i on similar tasks

$\|\mathbf{m}_i - \mathbf{m}_j\|_1$

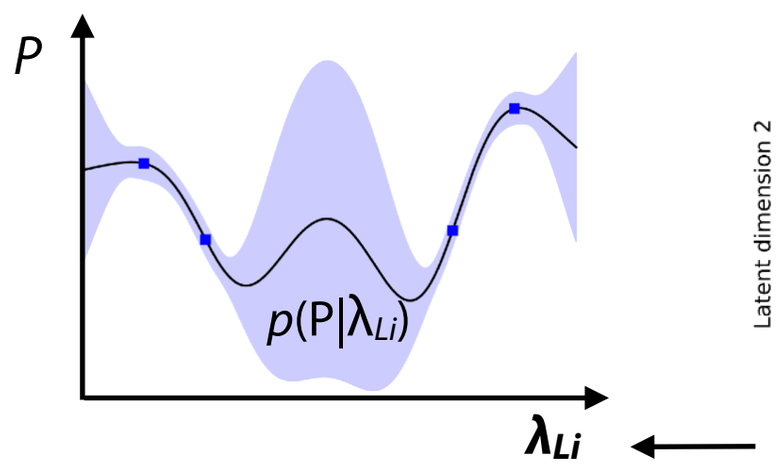


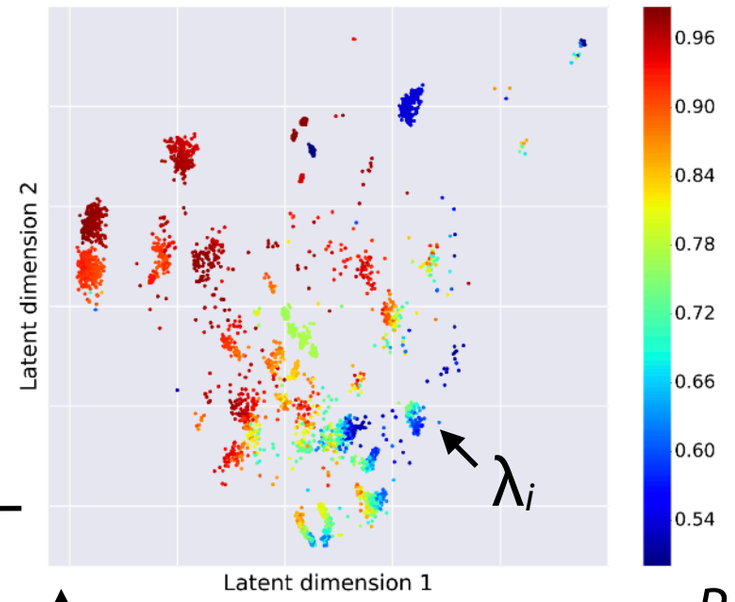


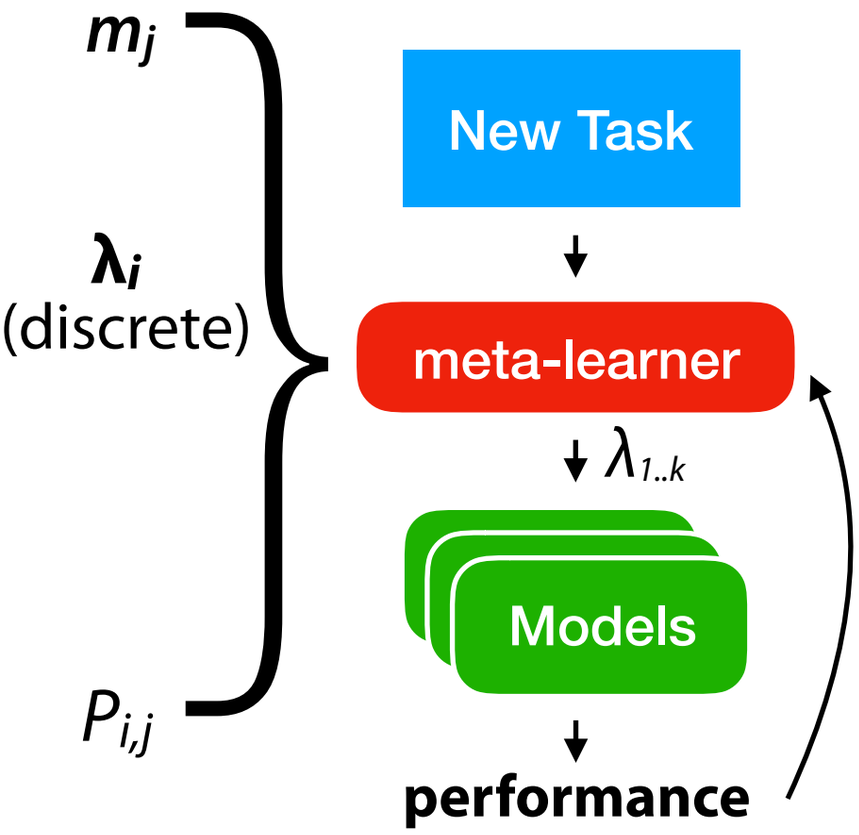
Warm-starting from similar tasks

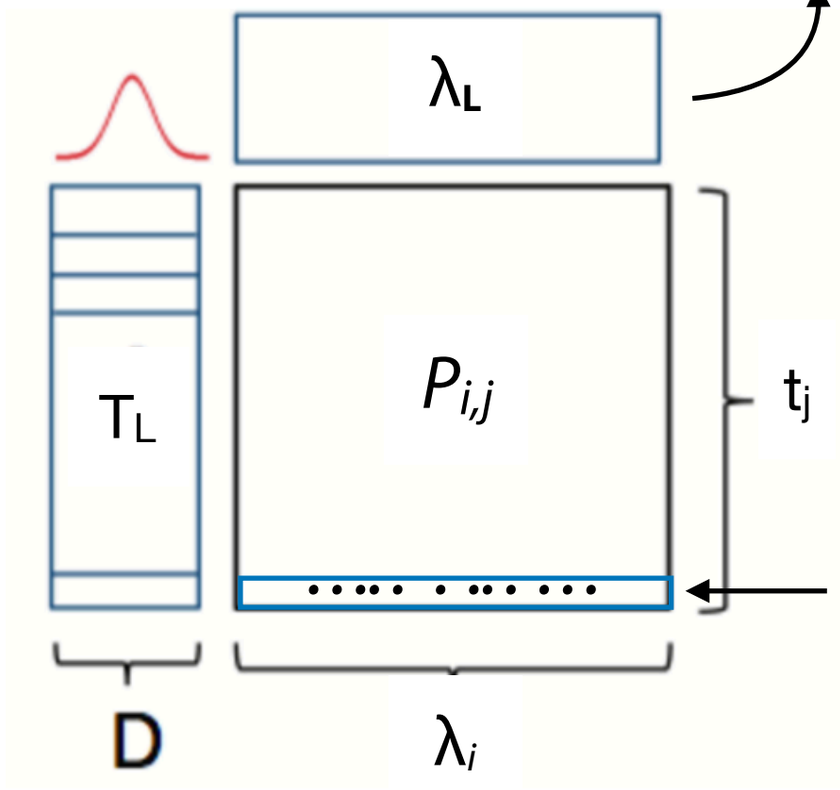
- Collaborative filtering: configurations λ_i are `rated' by tasks t_j

- Learn latent representation for tasks and configurations
- Use meta-features to warm-start on new task
- Returns probabilistic predictions for BayesOpt









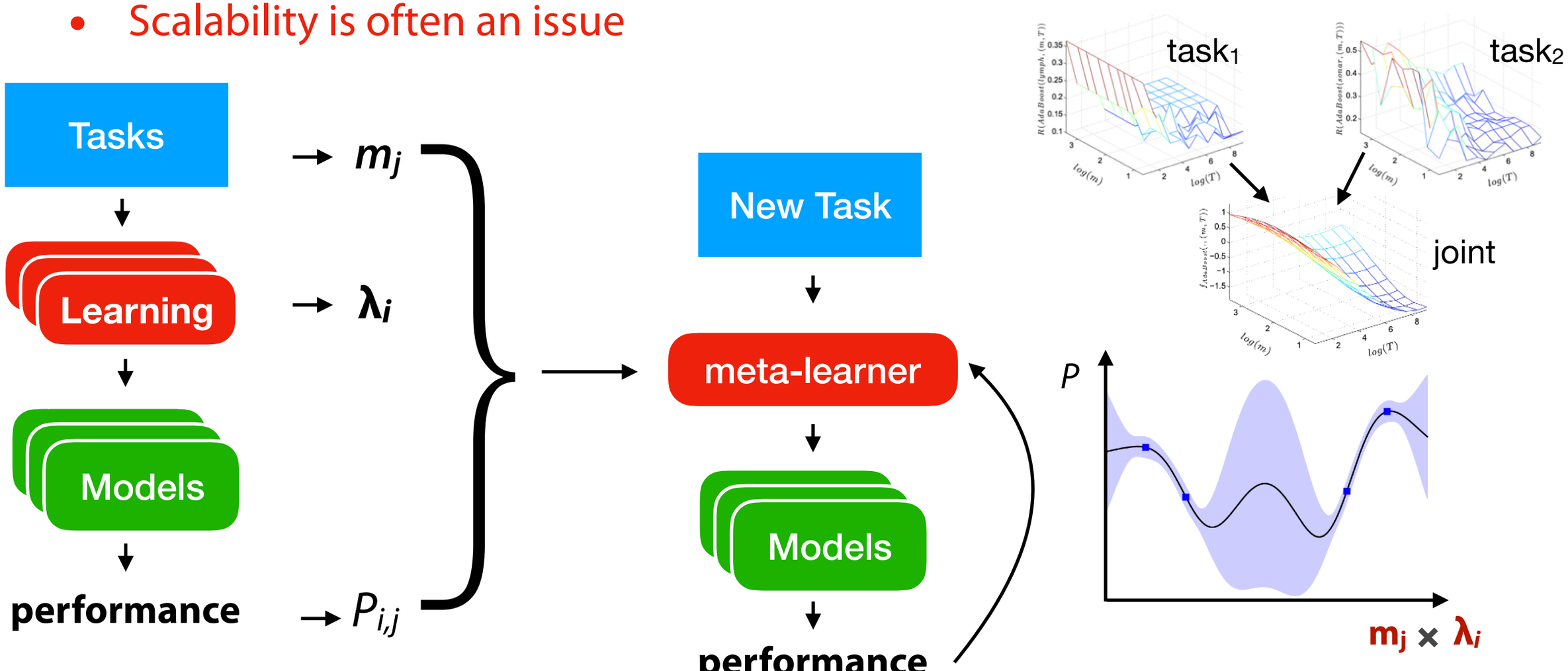
latent representation

t_{new} warm-started with $\lambda_{1..k}$

[1] Bardenet et al. 2013
[2] Schilling et al. 2015
[3] Yogatama et al. 2014

Global surrogate models

- Train a task-independent surrogate model with meta-features in inputs
  - SCOT: Predict *ranking* of λ_i with surrogate ranking model + m_j. [1]
  - Predict $P_{i,j}$ with multilayer Perceptron surrogates + m_j. [2]
  - Build joint GP surrogate model on most similar ($\|\mathbf{m}_i - \mathbf{m}_j\|_2$) tasks. [3]
  - Scalability is often an issue



[1] Brazdil et al. 2009, Lemke et al. 2015

[2] Sun and Pfahringer 2013, Pinto et al. 2017

[3] Sanders and C. Giraud-Carrier 2017

[4] Yang et al. 2018

Meta-models

- *Learn* direct mapping between meta-features and $P_{i,j}$
  - Zero-shot meta-models: predict best λ_i given meta-features [1]

  - Ranking models: return ranking $\lambda_{1..k}$ [2]

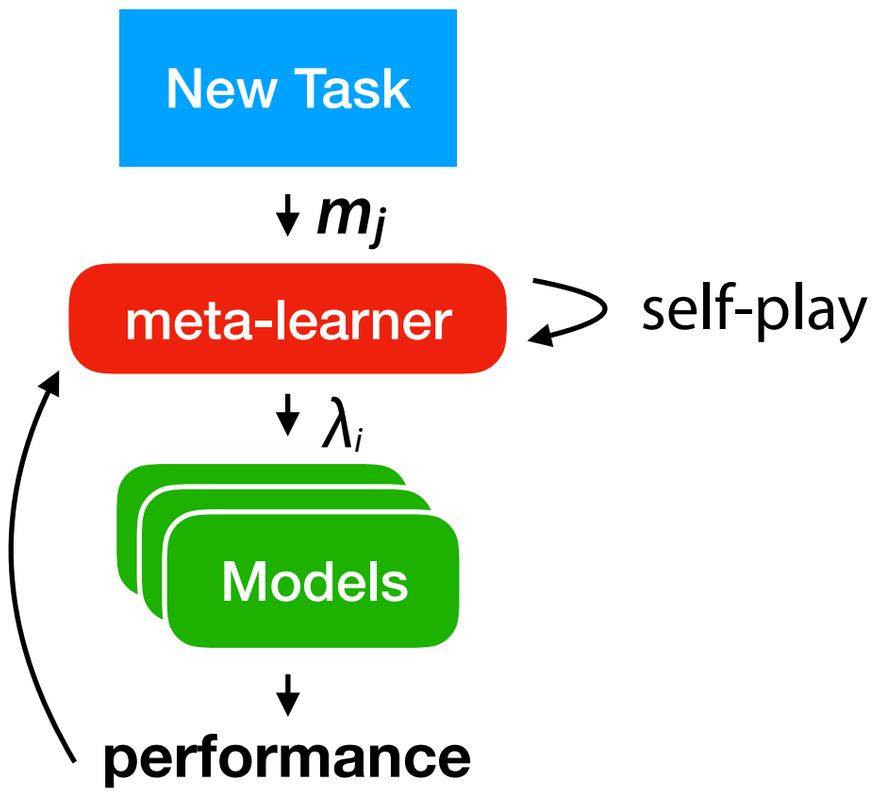
  - Predict which algorithms / configurations to consider / tune [3]

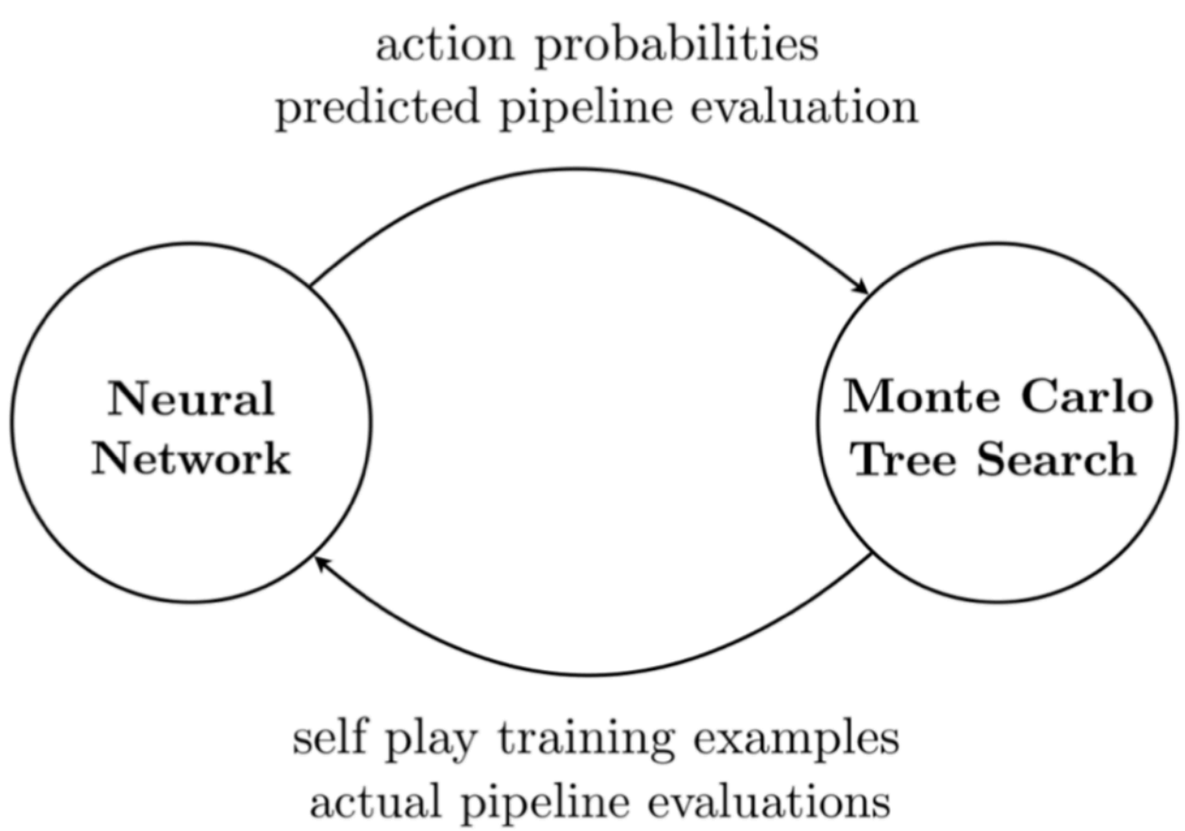
  - Predict performance / runtime for given θ_i and task [4]

- Can be integrated in larger AutoML systems: warm start, guide search,…

Learning to learn through self-play

- Build pipelines by inserting, deleting, replacing components (actions)
- Neural network (LSTM) receives task meta-features, pipelines and evaluations
  - Predicts pipeline performance and action probabilities
- Monte Carlo Tree Search builds pipelines, runs simulations against LSTM

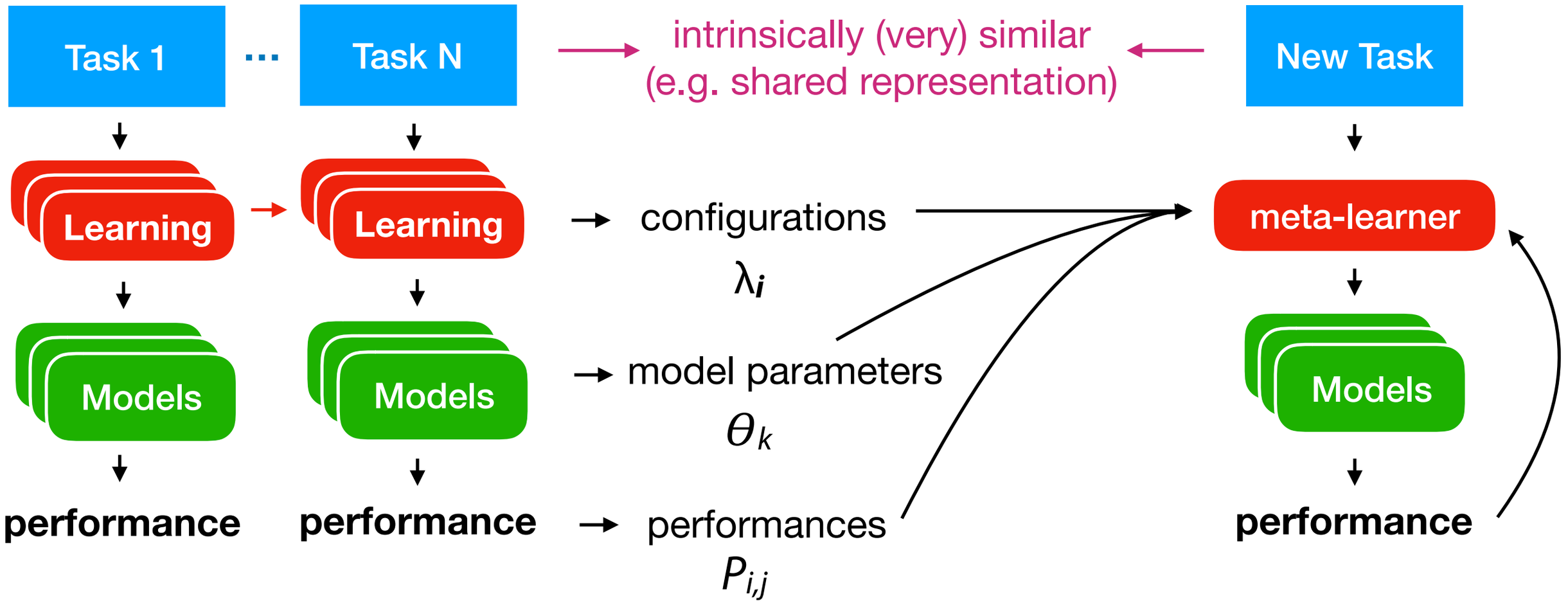




3. Learn from previous *models* (for *very similar* tasks)

Models trained on intrinsically similar tasks

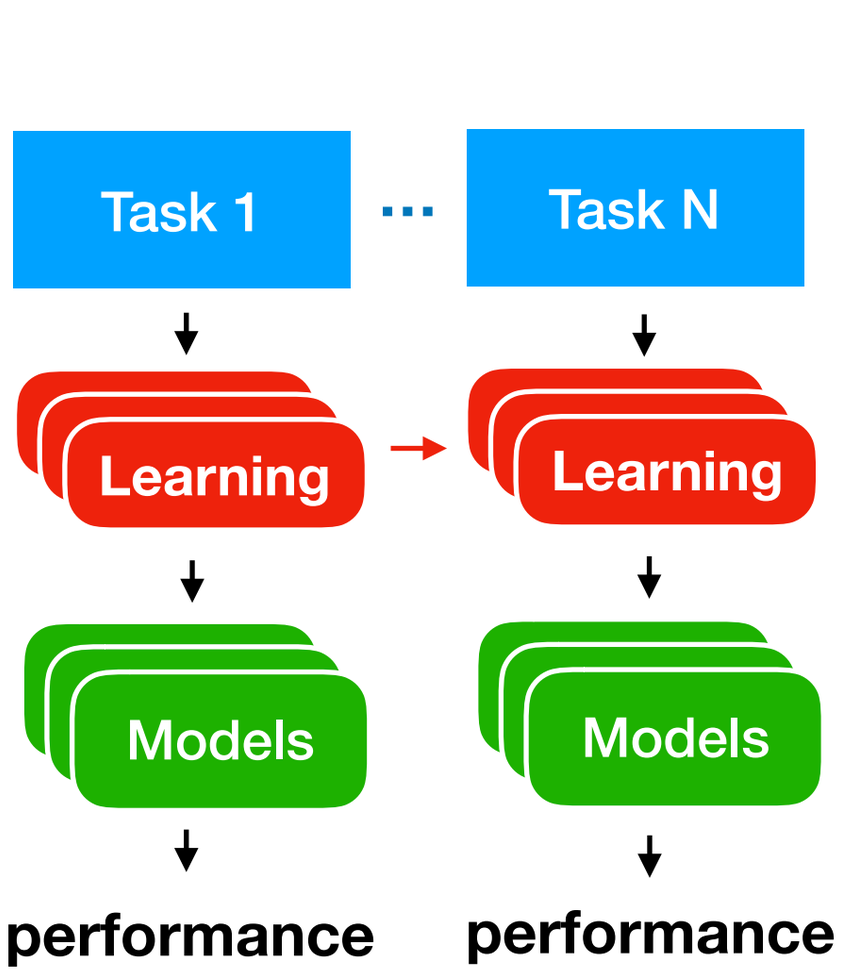
(model parameters, features,…)



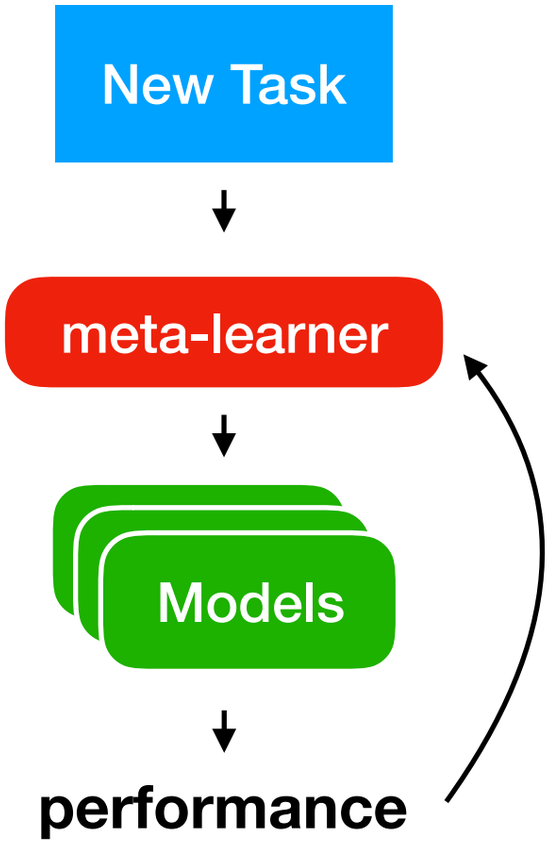
3. Learn from previous *models* (for *very similar* tasks)

Models trained on intrinsically similar tasks

(model parameters, features,…)



model param θ	model config λ	score P
0.34,0.43,…	alg=NN, lr=0.9	0,876
0.04,0.03,…	alg=NN, lr=0.9	0,921
0.94,0.03,…	alg=NN, lr=0.9	0,936
0.34,0.23,…	alg=NN, lr=0.5	0,674
0.04,0.23,…	alg=NN, lr=0.5	0,777
0.94,0.23,…	alg=NN, lr=0.5	0,791



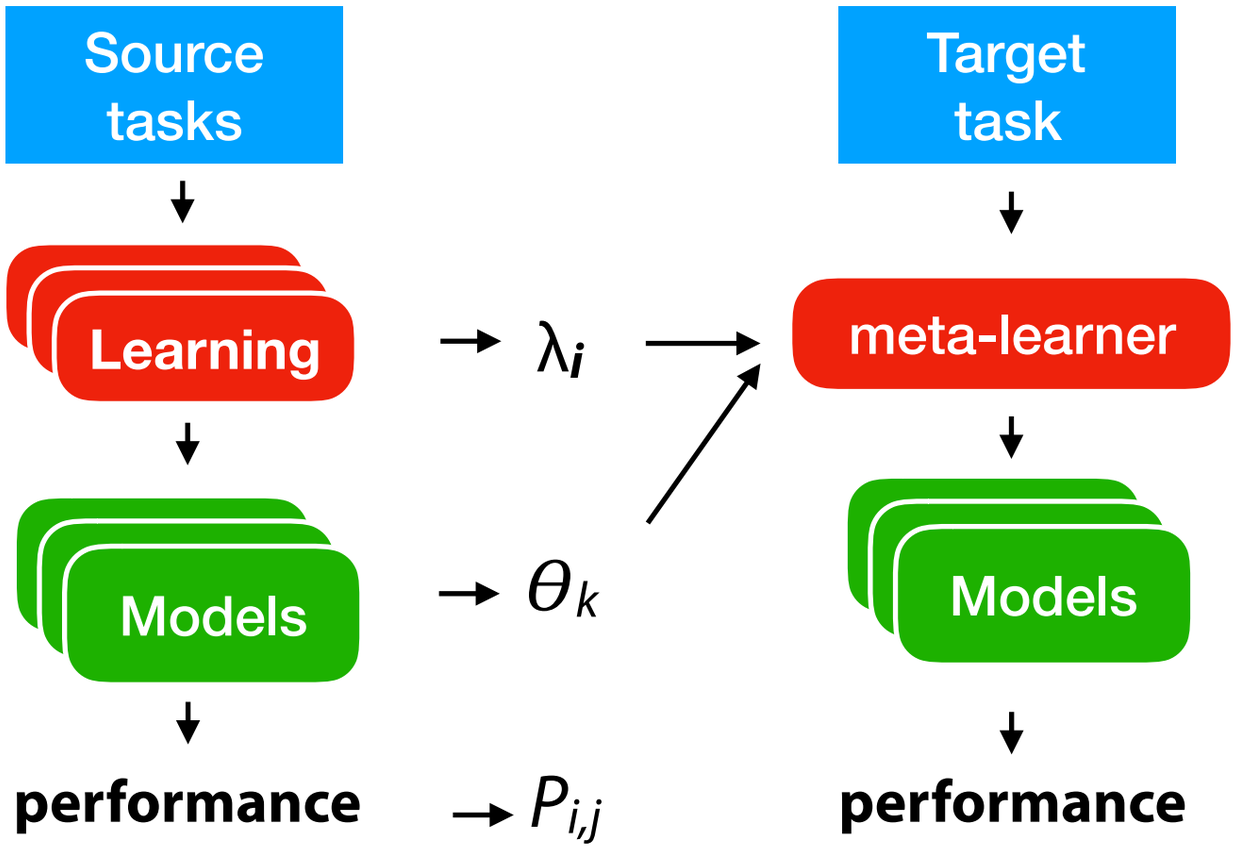
[1] Thrun and Pratt 1998

[2] Niculescu-Mizil and Caruana 2005

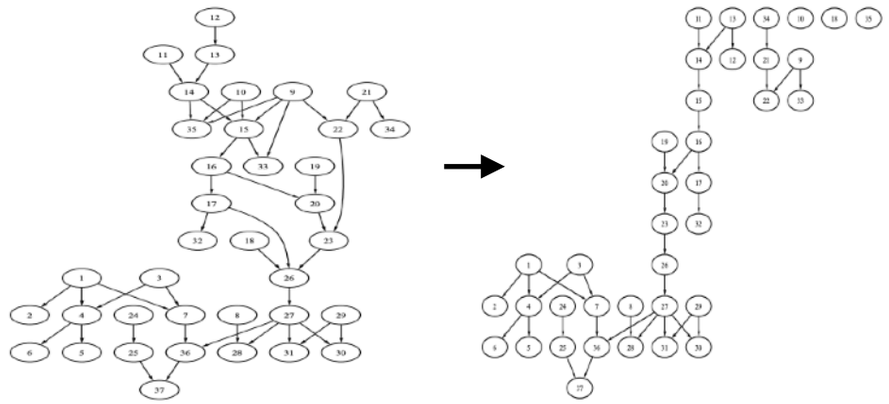
[3] Taylor and Stone 2009

Transfer Learning

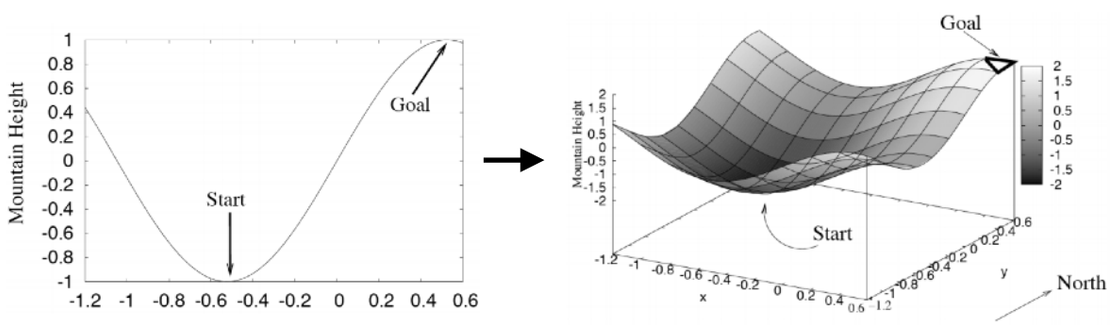
- Select source tasks, transfer trained models to similar target task [1]
- Use as starting point for tuning, or *freeze* certain aspects (e.g. structure)
  - Bayesian networks: start structure search from prior model [2]
  - Reinforcement learning: start policy search from prior policy [3]



Bayesian Network transfer



Reinforcement learning: 2D to 3D mountain car

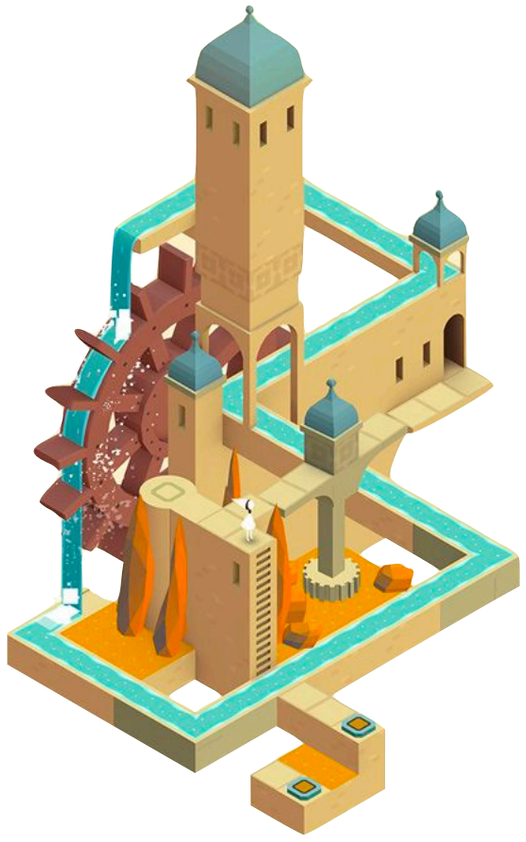


Learning to learn Neural networks

- End-to-end differentiable models afford many meta-learning opportunities
  - Transfer learning
  - Learning gradient descent procedures (weight update rules)
  - Few shot learning
  - Model-agnostic meta-learning (learning weight initializations)
  - Learning to reinforcement learn
  - …

See part 3

Advanced Course on Data Science & Machine Learning
Siena, 2019



Automatic Machine Learning & Meta-Learning

part 3

Joaquin Vanschoren

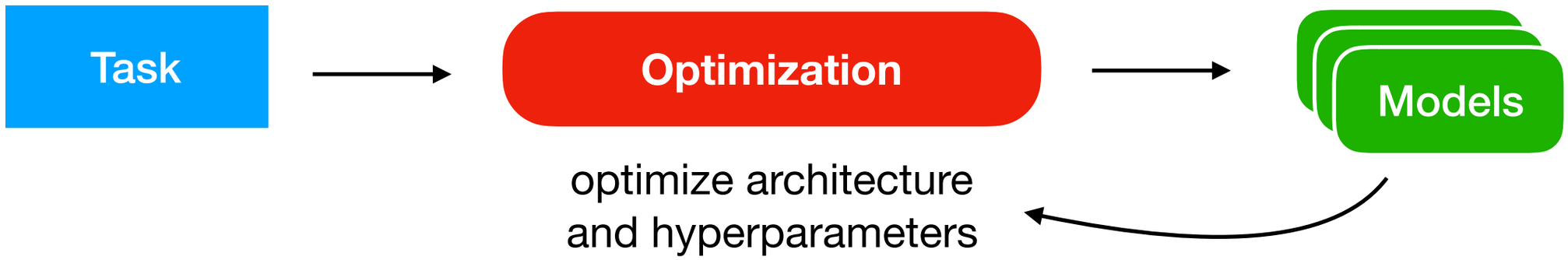
Eindhoven University of Technology

j.vanschoren@tue.nl @joavanschoren http://bit.ly/openml

slides!

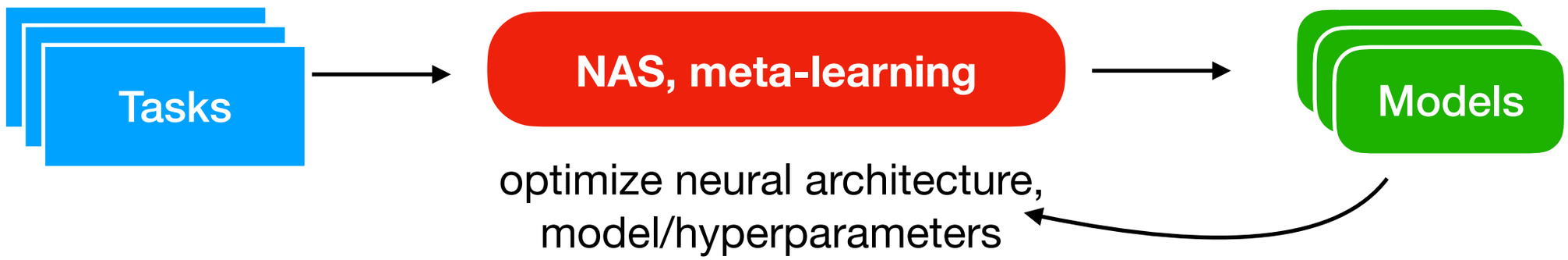
Overview

part 1 AutoML introduction, promising optimization techniques



part 2 Meta-learning

part 3 Neural architecture search, meta-learning on neural nets



Overview

part 3 Neural architecture search, meta-learning on neural nets

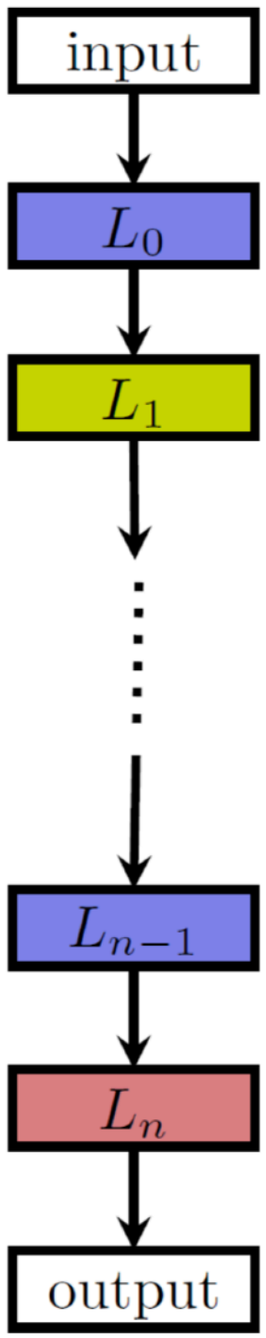
1. Neural Architecture Search space
2. Neural Architecture optimization
3. (Neural) Meta-Learning

Neural Architecture Search space

Sequential (chain)

Choose:
- number of layers
- type of layers
  - dense
  - convolutional
  - max-pooling
  - …
- hyperparameters of layers

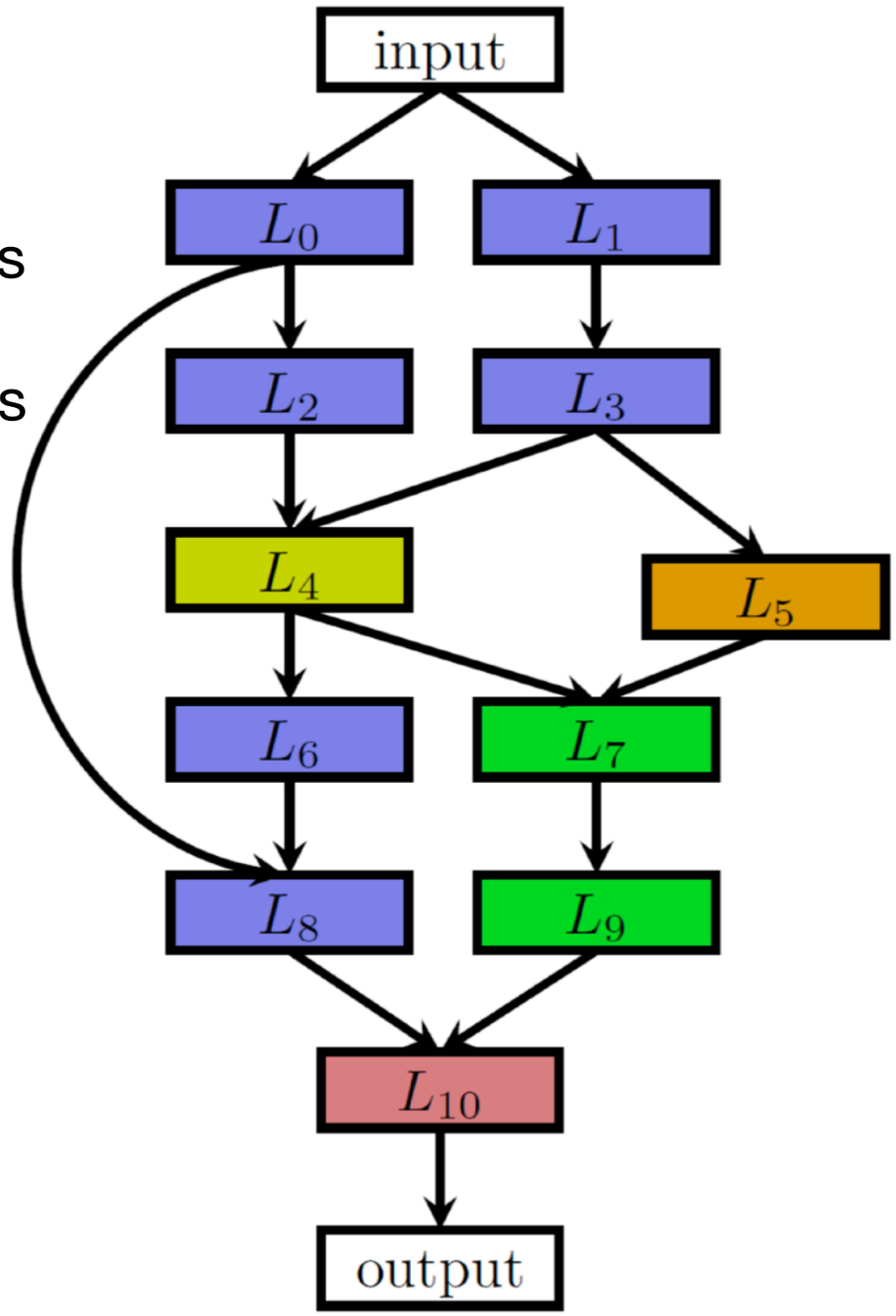
+ easier to search
- sometimes too simple



Arbitrary graph

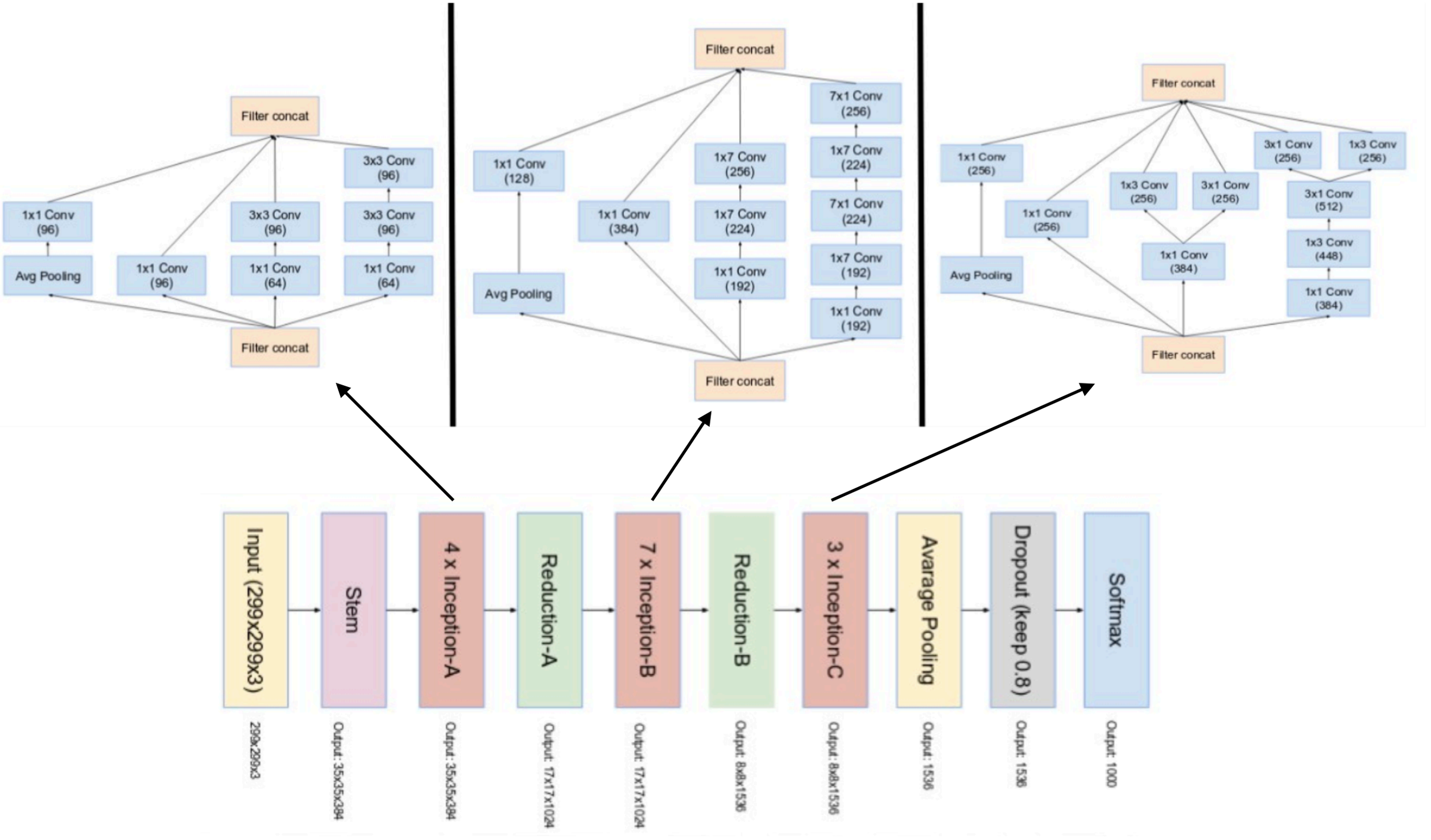
Choose:
- branching
- joins
- skip connections
- types of layers
- hyperparameters of layers

+ more flexible
- much harder to search



Neural Architecture Search space

Observation: successful deep networks have repeated motifs (cells)
e.g. Inception v4:



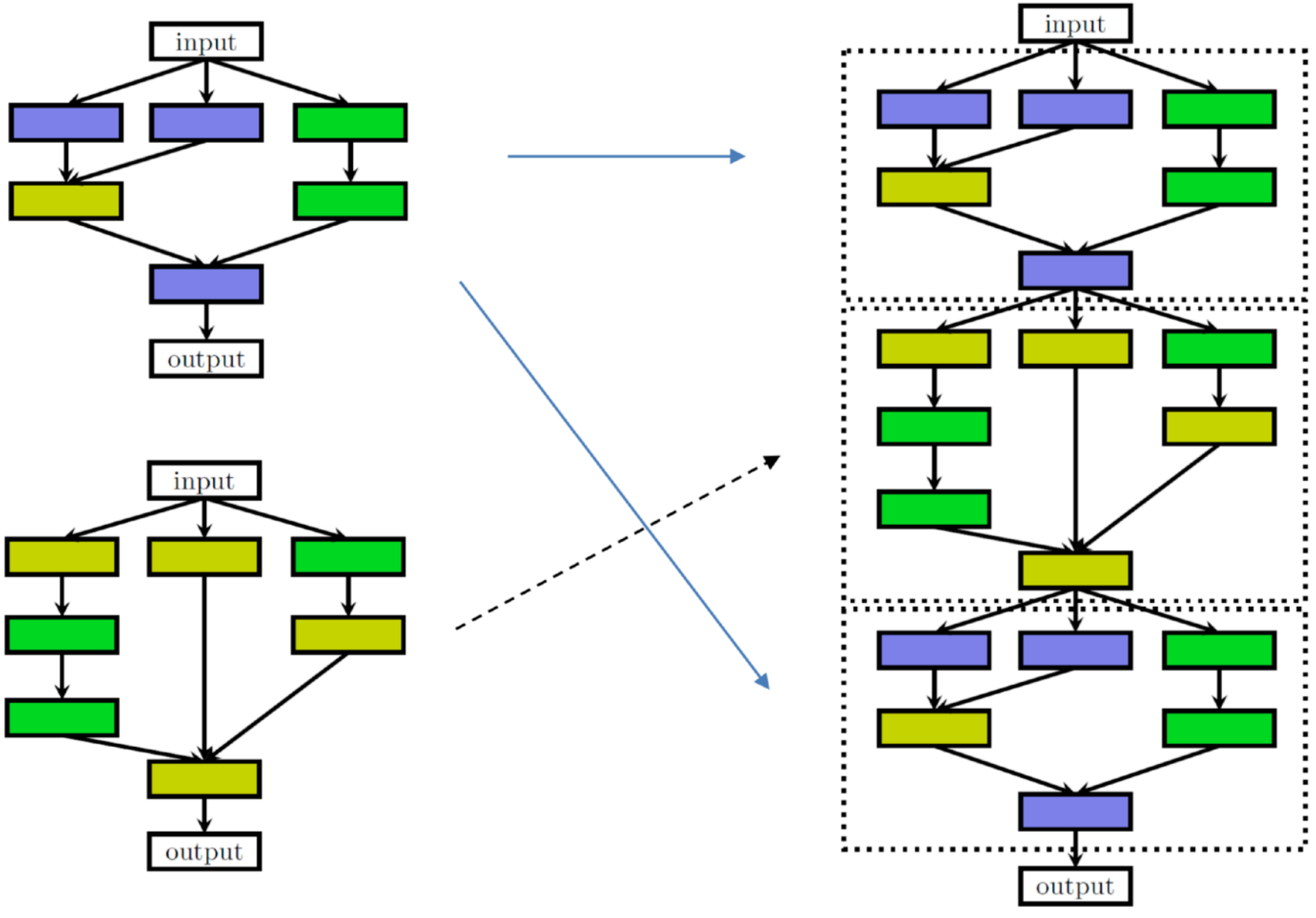
Zoph et al 2018

Cell search space

- parameterize building blocks *(cells)*
  - *e.g. regular cell + resolution reduction cell*

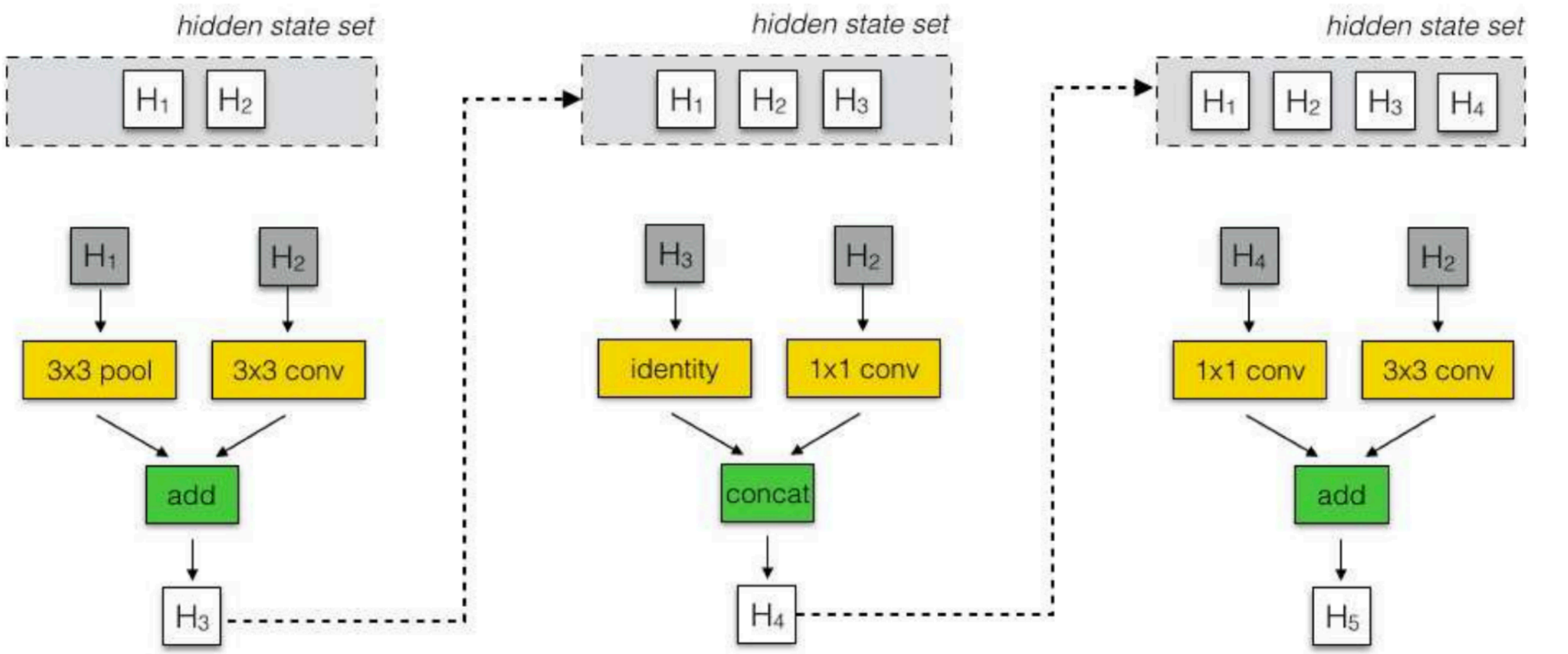
- stack cells together in macro-architecture
  - usually a chain
  - can be manual or learned

+ smaller search space
+ cells can be learned on a small dataset & transferred to a larger dataset
- you can't learn entirely new architectures



Within-cell search space

- Different *strategies* to construct cell, leading to different parameterizations
- Can be parameterized as set of categorical hyperparameters:
  - Which existing layer (hidden state, e.g. cell input) H_i to build on
  - Which operation (e.g. 3x3conv) to perform on H_i
  - How to combine into new hidden state
  - Iterate over *B* phases (blocks)

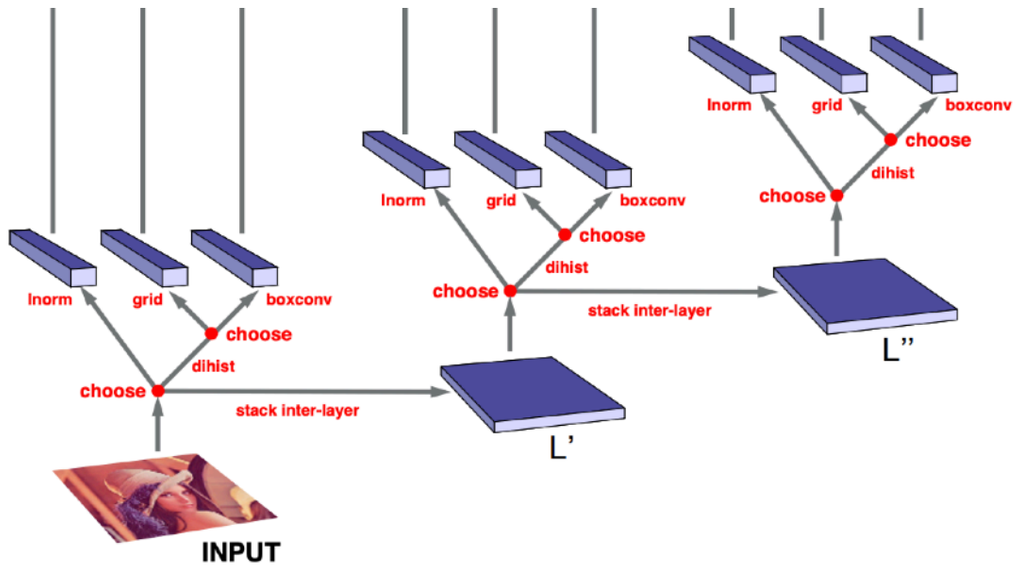


[1] Bergstra et al. 2013
[2] Swersky et al 2013
[3] Kandasamy et al. 2018

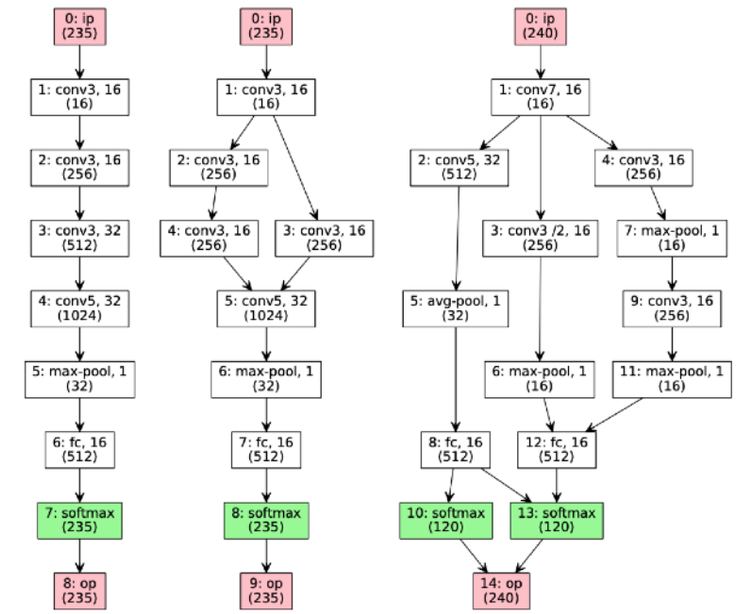
Neural architecture optimization

Standard hyperparameter optimization techniques

- Image classification pipelines[1]
  - 238 hyperparameters, tuned with TPE



- Kernels to optimize neural nets with GP-based Bayesian optimization
  - ArcNet[2]: DNNs, 23 hyperparameters, 6 kernels
  - NASBOT[3]: DNNs and CNNs



[1] Mendoza et al. 2016

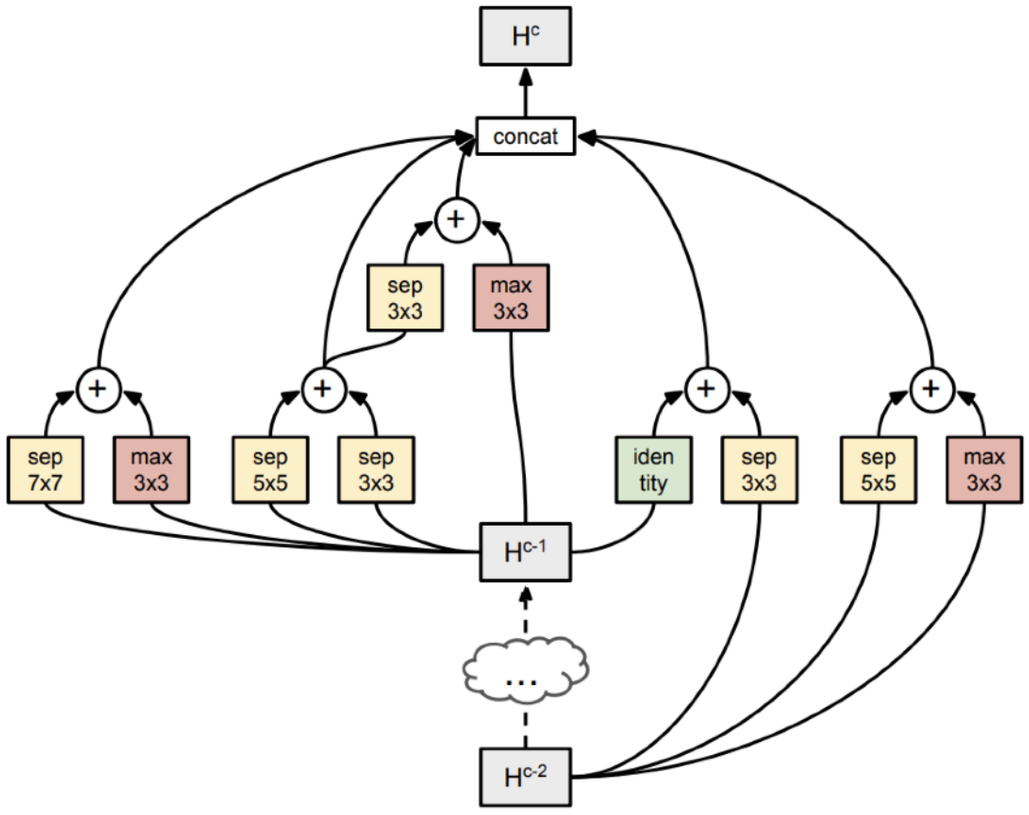
[2] Zela et al. 2018

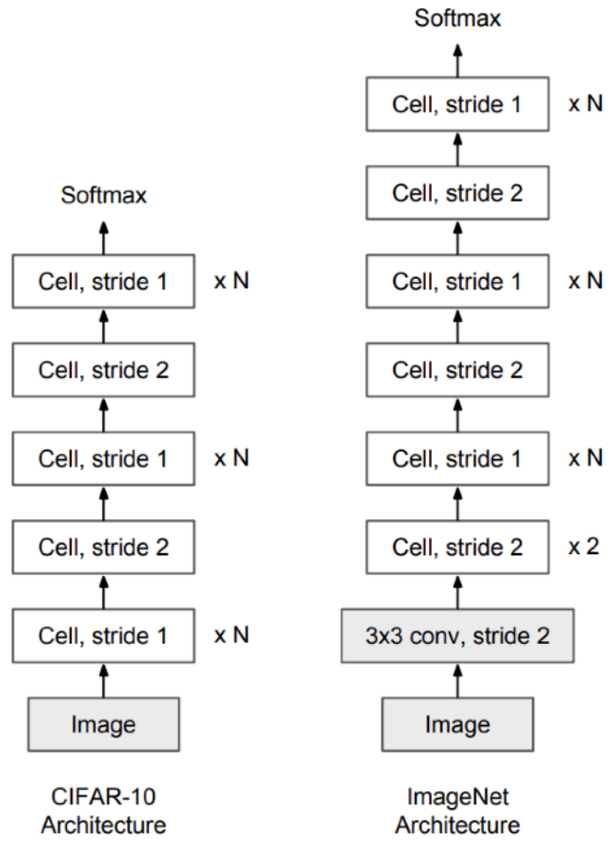
[3] Liu et al. 2018

Neural architecture optimization

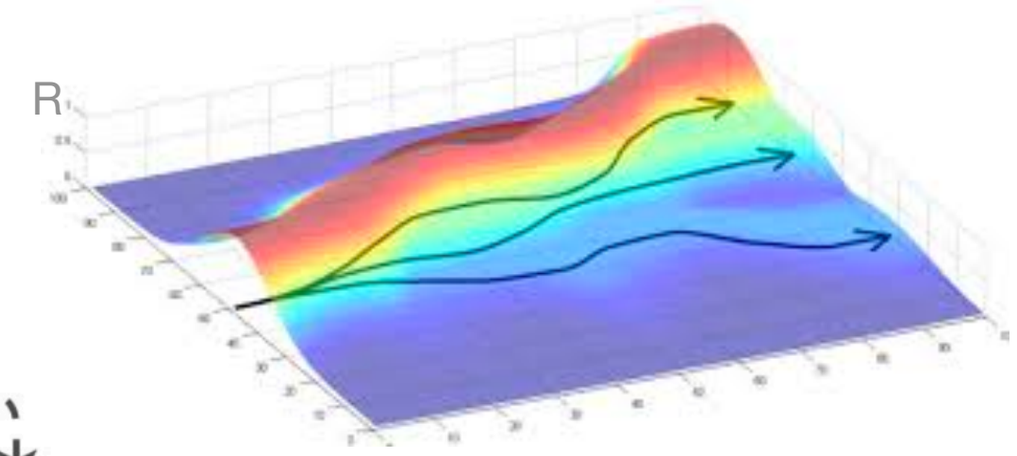
Standard hyperparameter optimization techniques

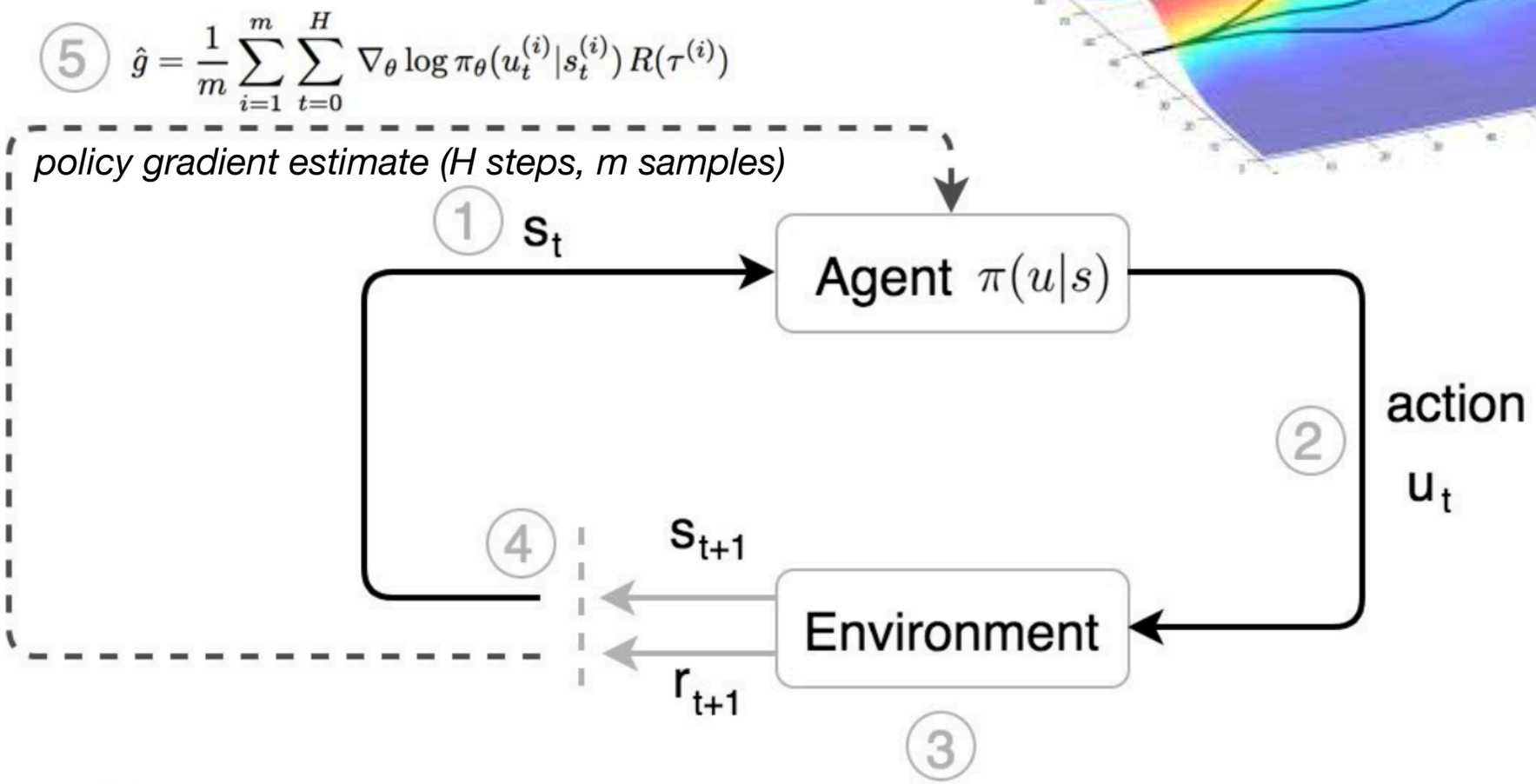
- AutoNet [1] : DNNs, 63 hyperparameters, tuned with SMAC
- Joint NAS + HPO [2]: ResNets, tuned with BO-HB
- PNAS (Progressive NAS) [3]
  - Cell search space, optimized with SMAC, HPO afterwards
  - SotA on ImageNet, CIFAR





Deep RL recap





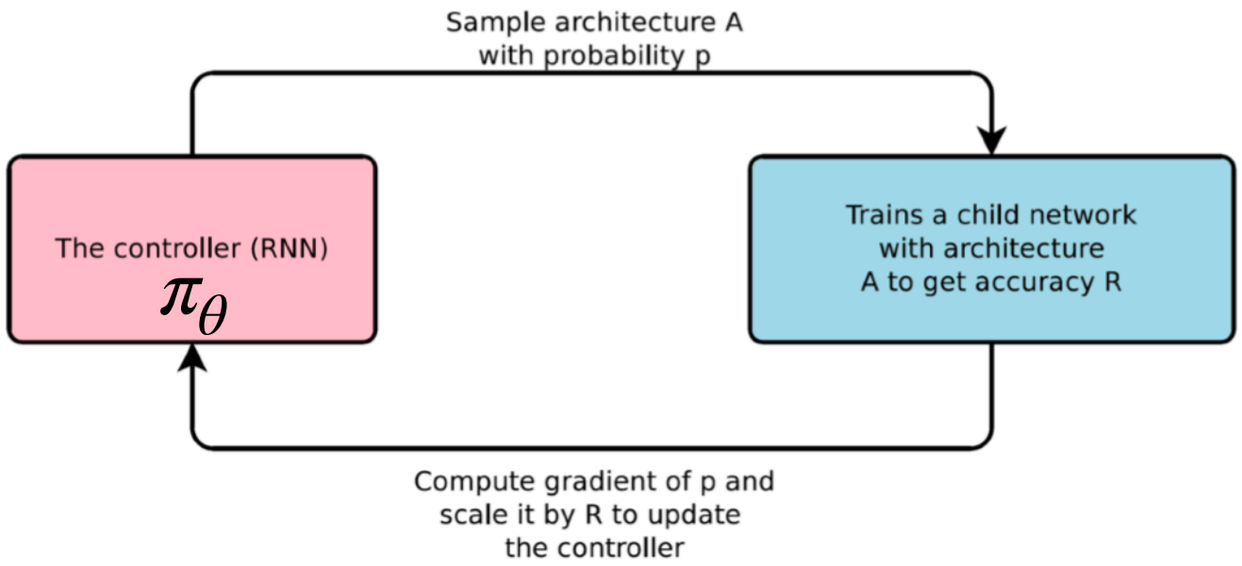
1. Observe state
2. Take action *u* based on policy π_θ (θ are weights in an RNN)
3. New state is formed
4. Take further actions based on new state
5. After trajectory τ of motions, adjust policy based on total reward R(τ)

Zoph & Le 2017

NAS with Reinforcement learning

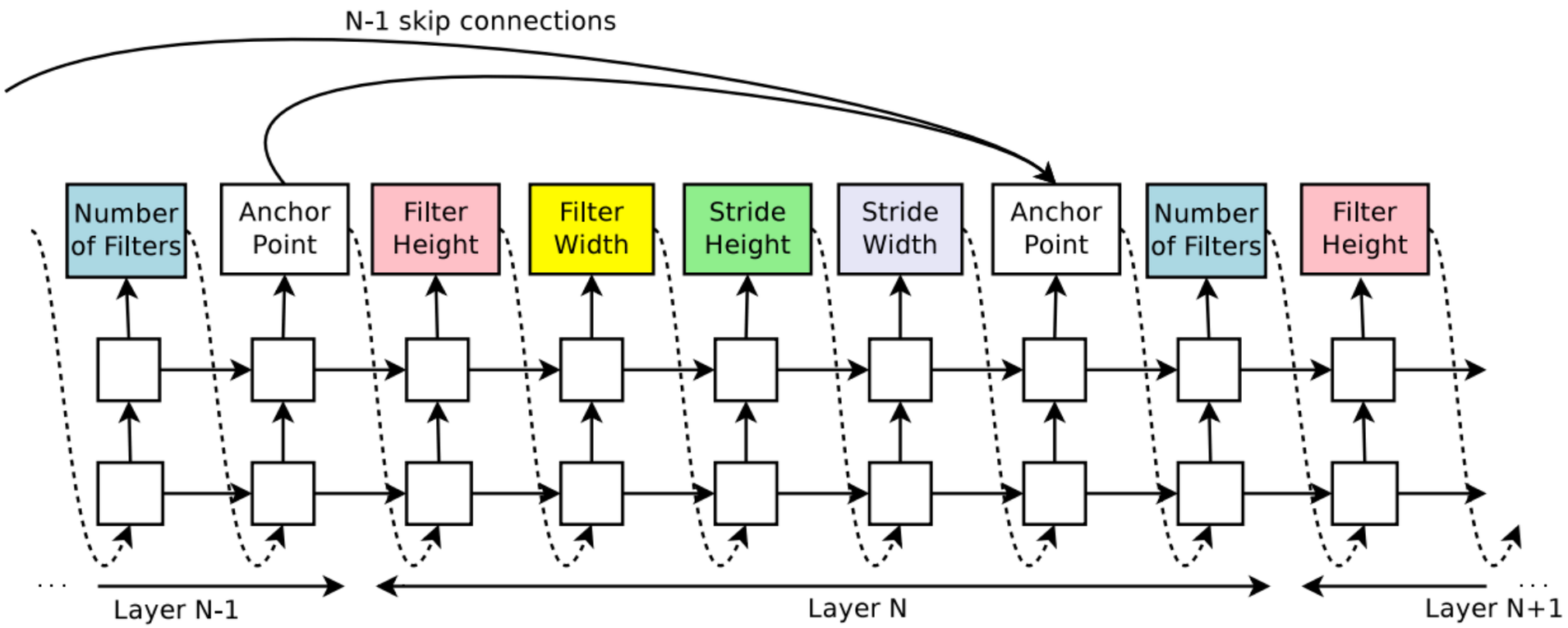
RNN policy network:

- generate architecture step by step
- evaluate to get reward
- update with policy gradient



2-layer LSTM (REINFORCE), chains of convolutional layers

- State of the art on CIFAR-10, Penn Treebank
- 800 GPUs for 3-4 weeks, 12800 architectures

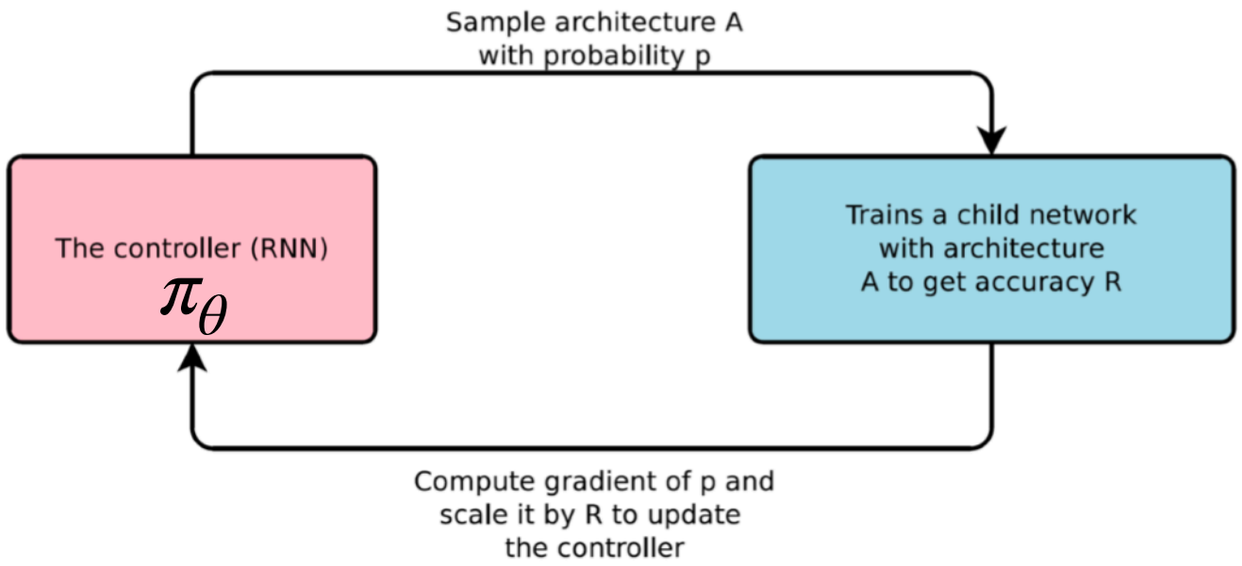


Zoph et al 2018

NAS with Reinforcement learning

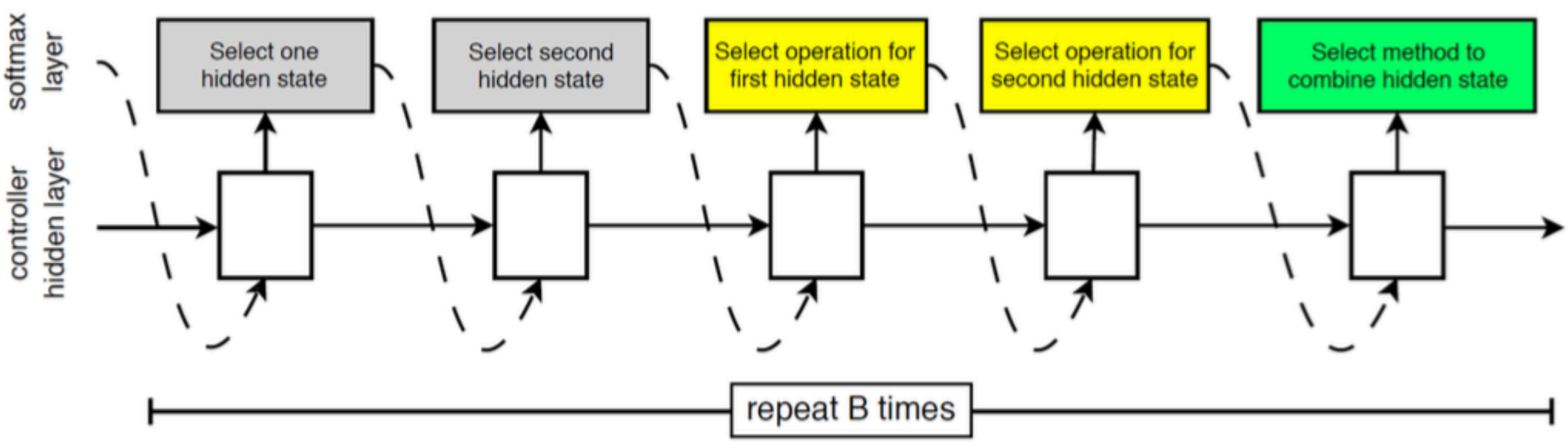
RNN policy network:

- generate architecture step by step
- evaluate to get reward
- update with policy gradient



1-layer LSTM (PPO), cell space search

- State of the art on ImageNet
- 450 GPUs, 20000 architectures

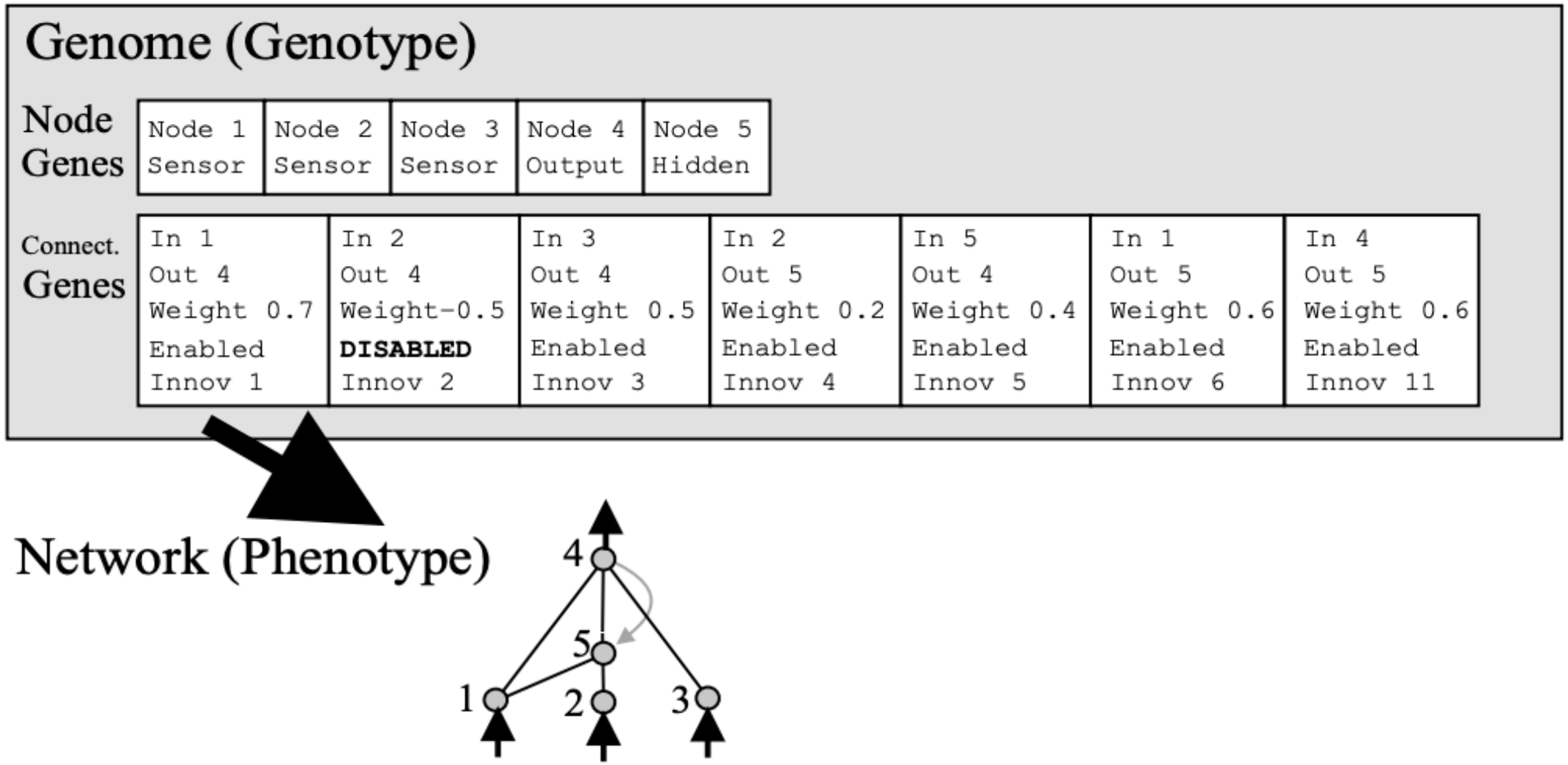


Angeline et al 1994, *Stanley and Miikkulainen 2002*

Neuroevolution

Earlier solutions tried to:

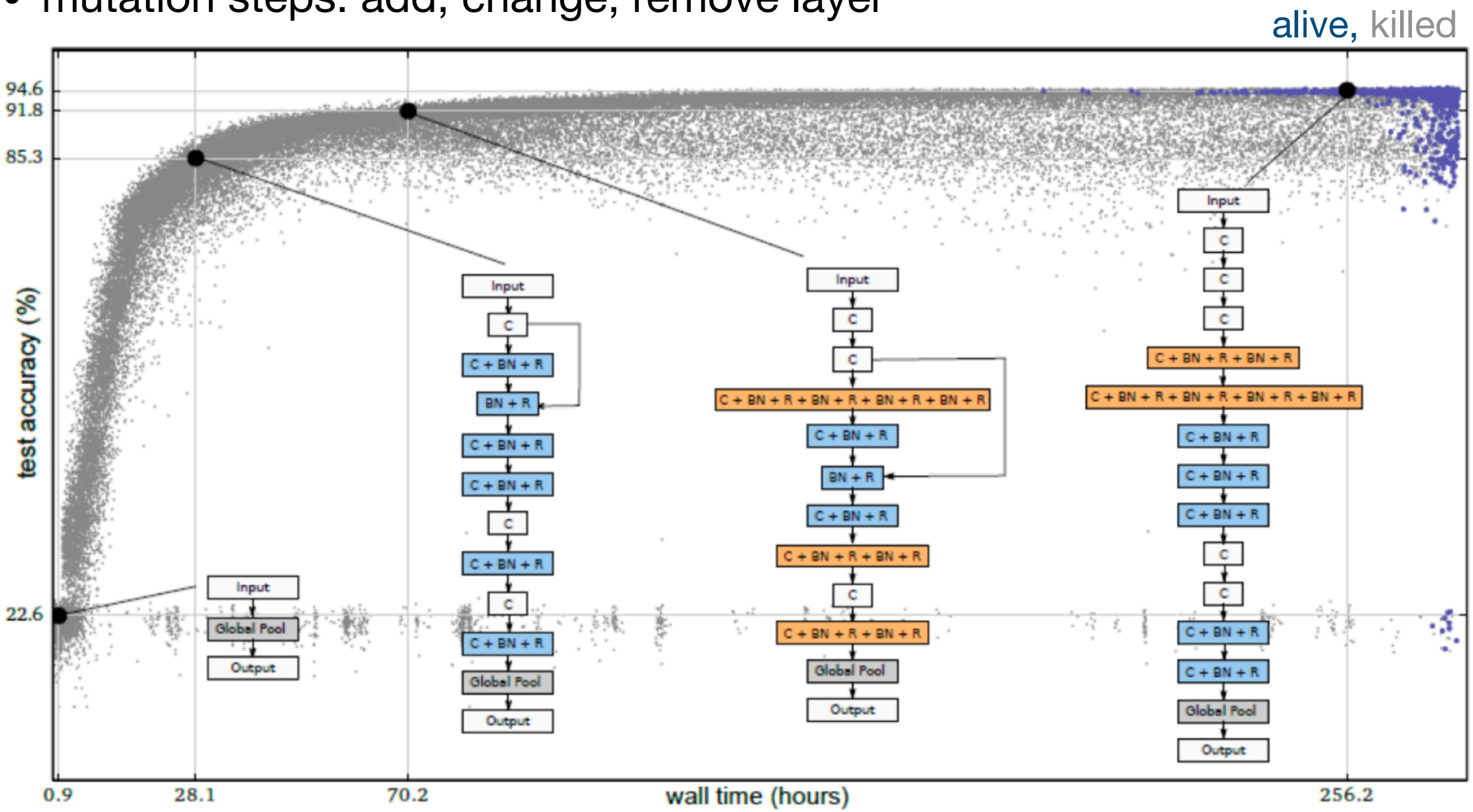
- Optimize both the configuration and the weights simultaneously with genetic algorithms
- Optimize individual nodes and connections
- Doesn't scale to deep networks



Neuroevolution

Better:

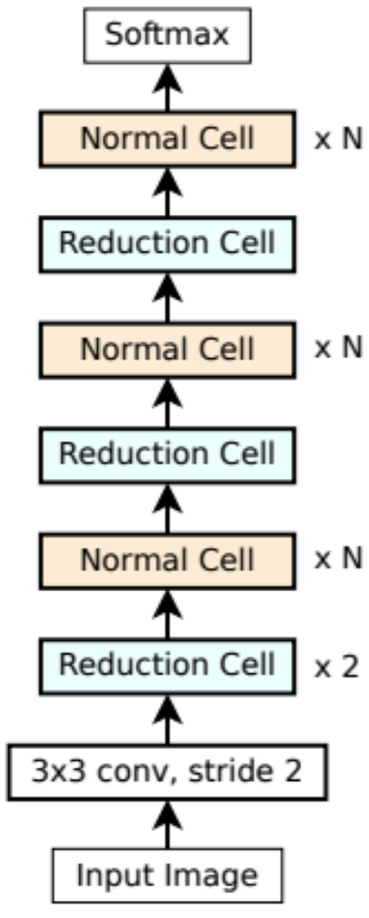
- learn the architecture and configuration with evolutionary techniques
- train the network with SGD
- mutation steps: add, change, remove layer

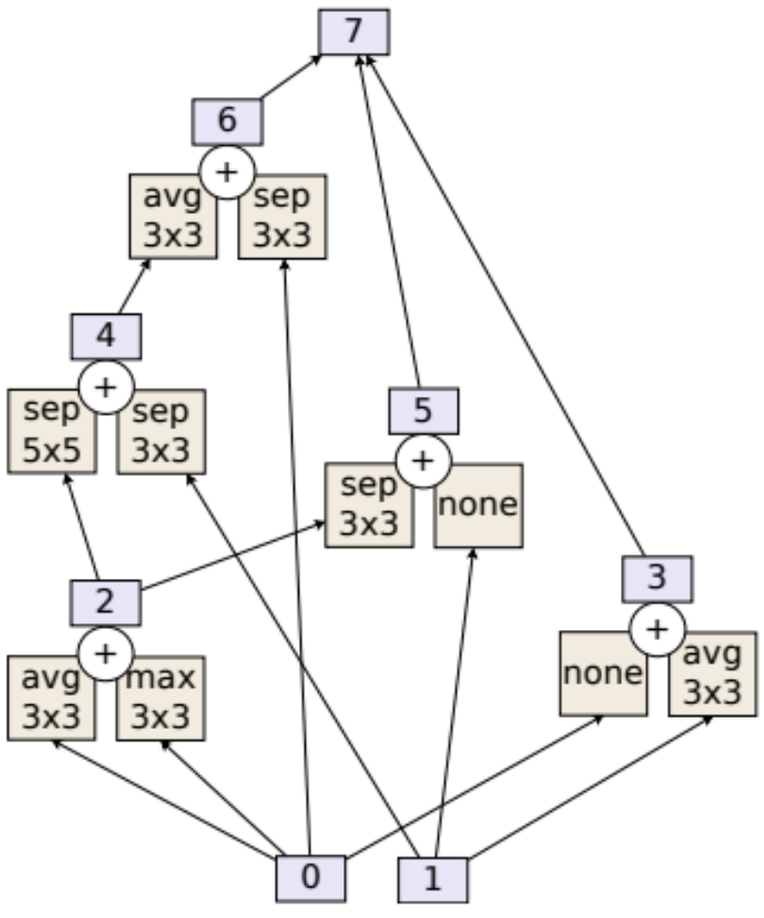


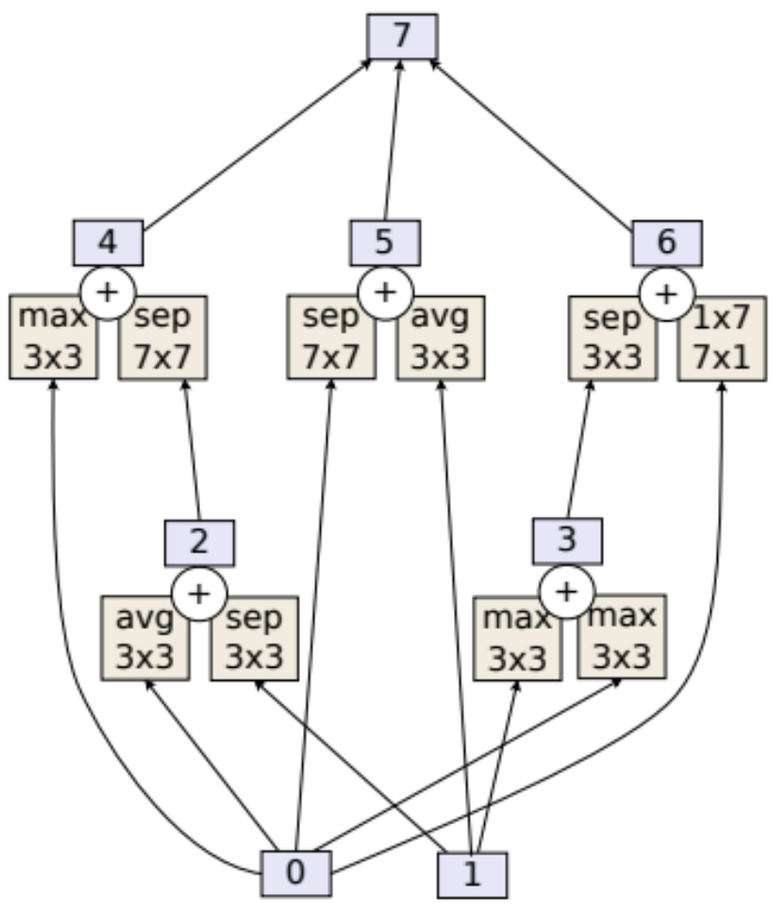
Neuroevolution

AmoebaNet: State of the art on ImageNet, CIFAR-10

- Cell search space, aging evolution (kill oldest networks)





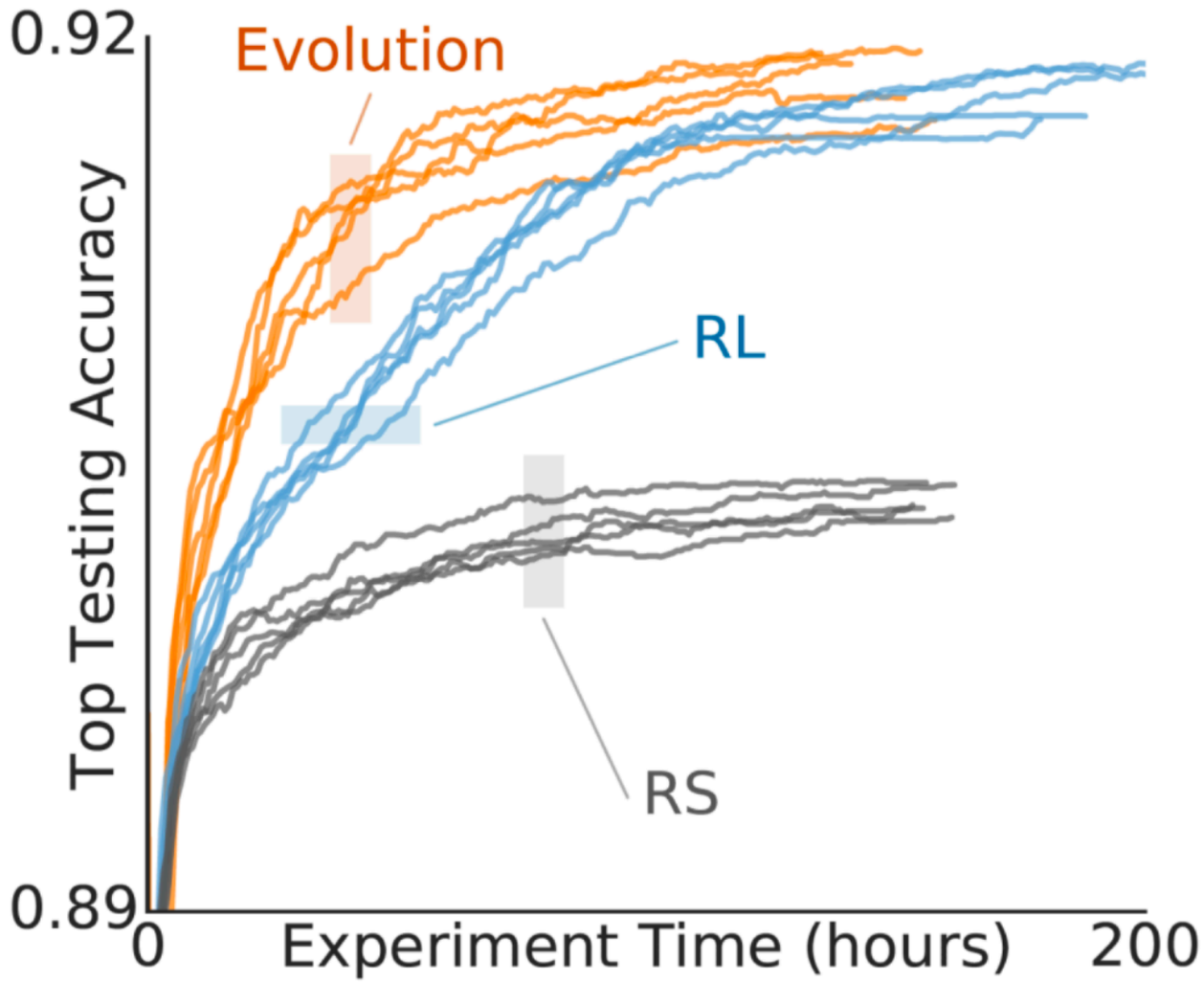


[Real et al 2019]

Neuroevolution

AmoebaNet: State of the art on ImageNet, CIFAR-10

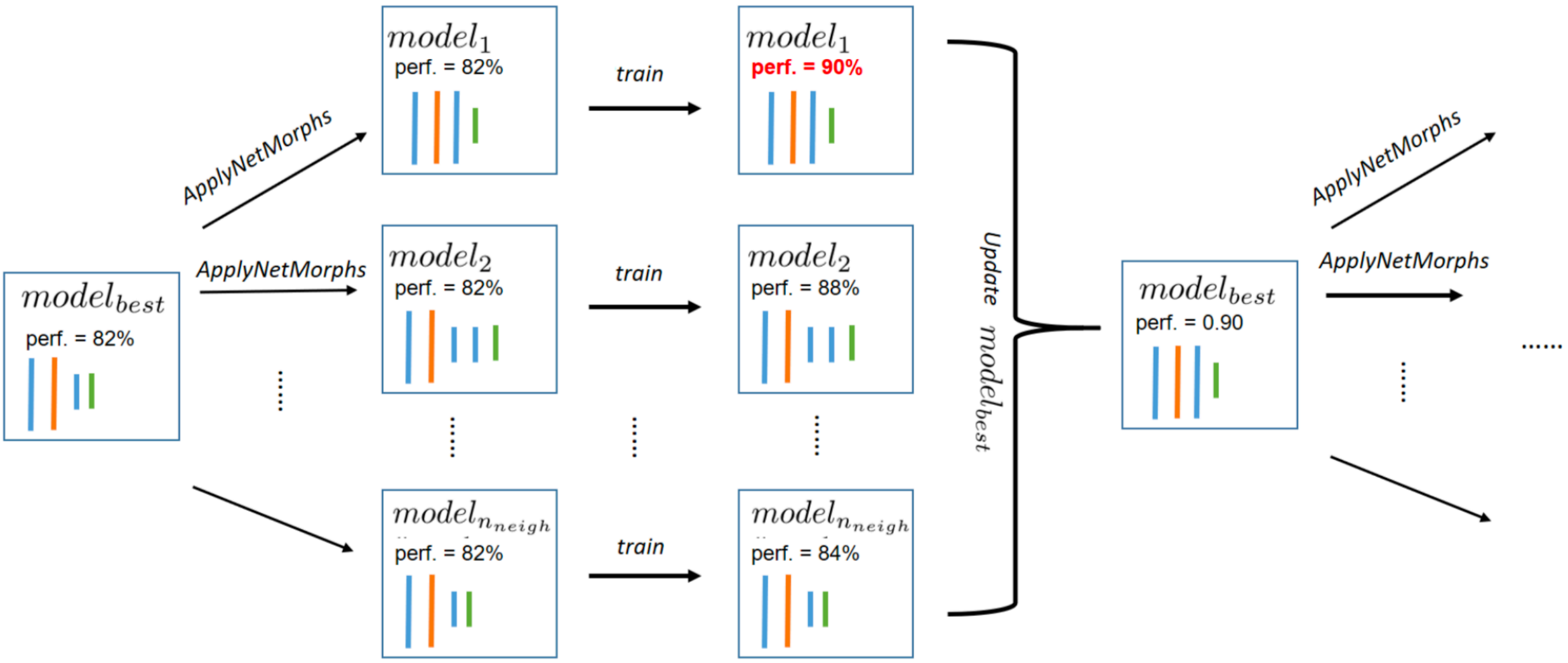
- Cell search space, aging evolution (kill oldest networks)
- More efficient than reinforcement learning



[Chen et al. 2016](), [Wei et al. 2016](), [Cai et al. 2017]()
[Cai et al. 2018](), [Elsken et al. 2017](), [Cortes et al. 2017](), [Cai et al. 2018]()

Network morphisms

- Change architecture, but not modelled function
- Non-black box, allows much more efficient architecture search



[1] Saxena & Verbeek, 2016

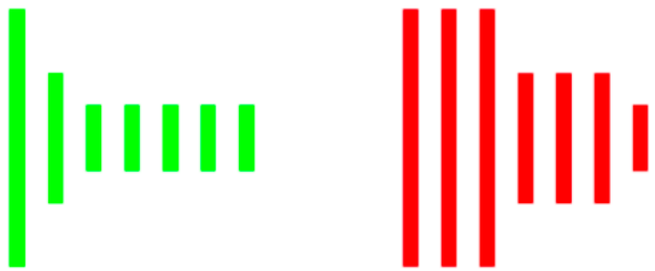
[2] Bender et al. 2018

[3] Pham et al. 2018

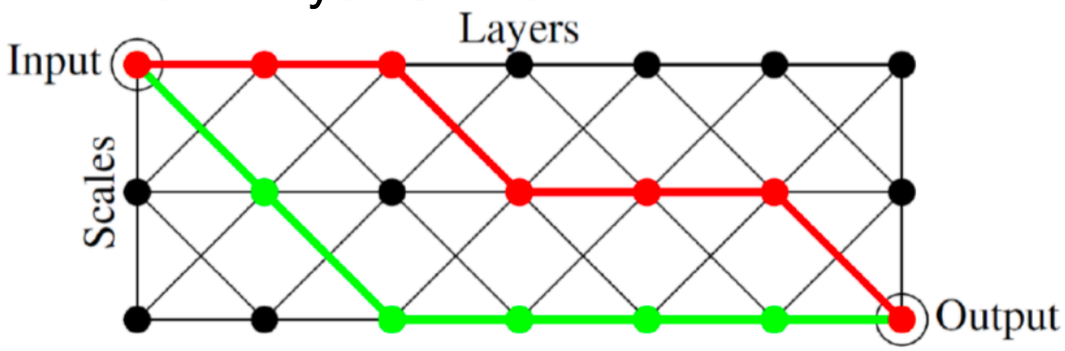
[4] Brock et al. 2017

Weight sharing

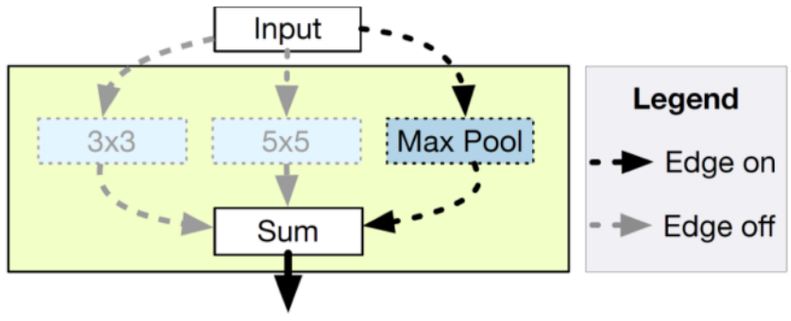
- One-shot models (convolutional neural fabric) [1]
  - Search space: ConvNet = path though *fabric* with shared weights
  - Train ensemble of all of them



For 7-layer CNNs:



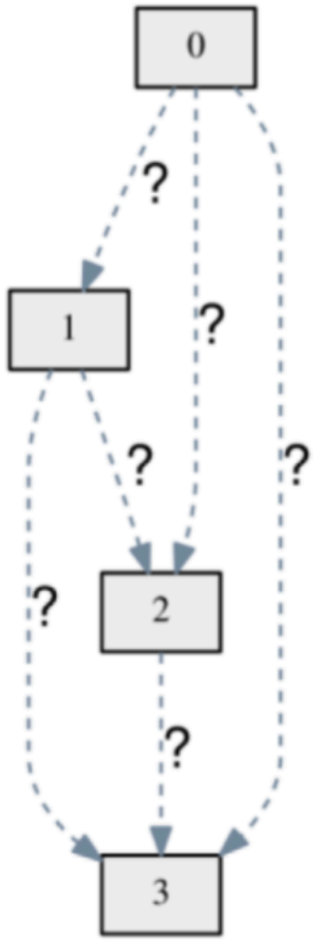
- *Path dropout* [2]
  - ensure that individual paths do well



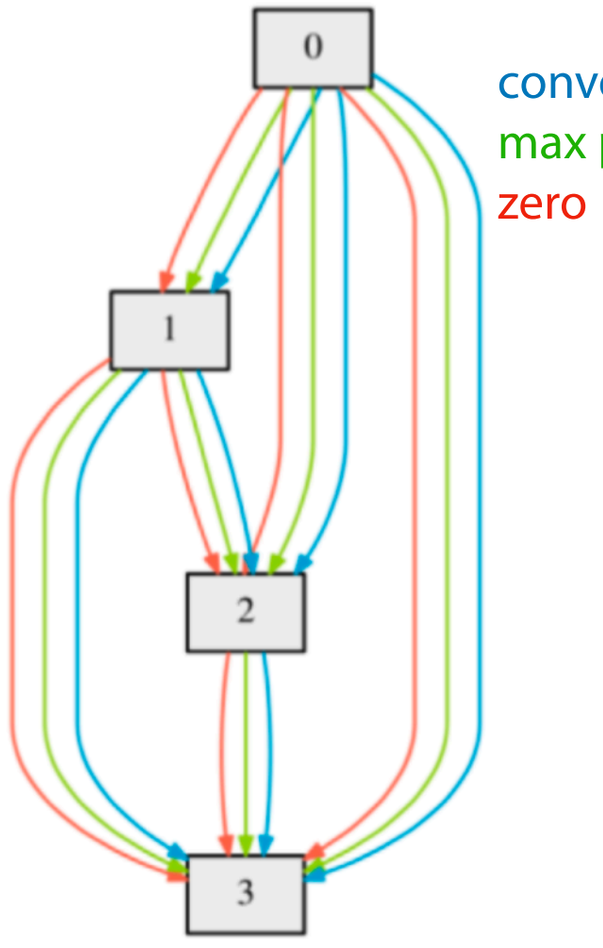
- *ENAS* [3]
  - use RL to sample paths from one-shot model, train weights
  - other paths inherit these weights
- *SMASH* [4]
  - Shared hypernetwork that predicts weights given architecture

DARTS: Differentiable NAS

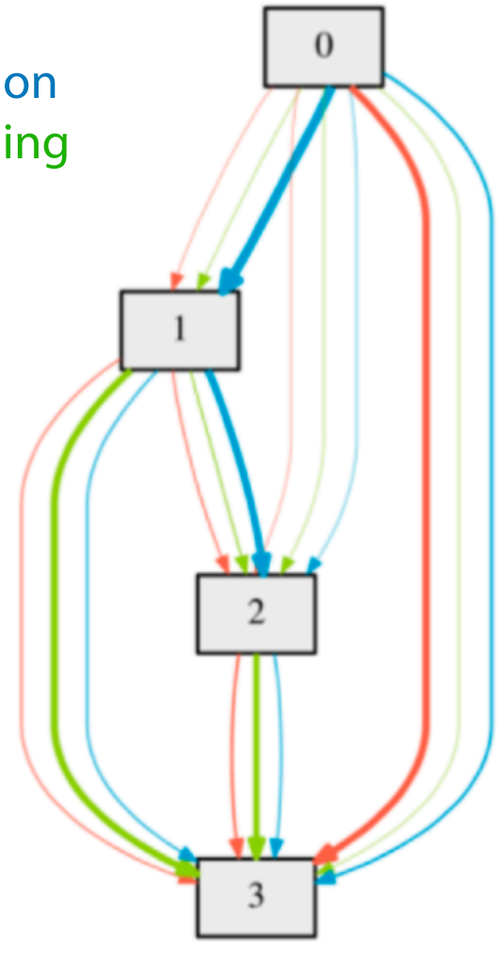
- Fixed (one-shot) structure, learn which operators to use
- Give all operators a weight α_i
- Optimize α_i and model weights ω_j using bilevel programming



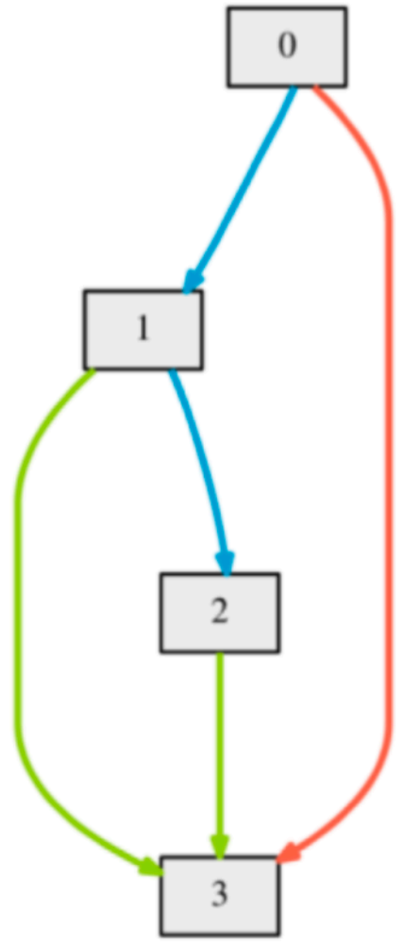
One-shot model



operator weights α_i



interleaved optimization of α_i and ω_j with SGD



argmax α_i

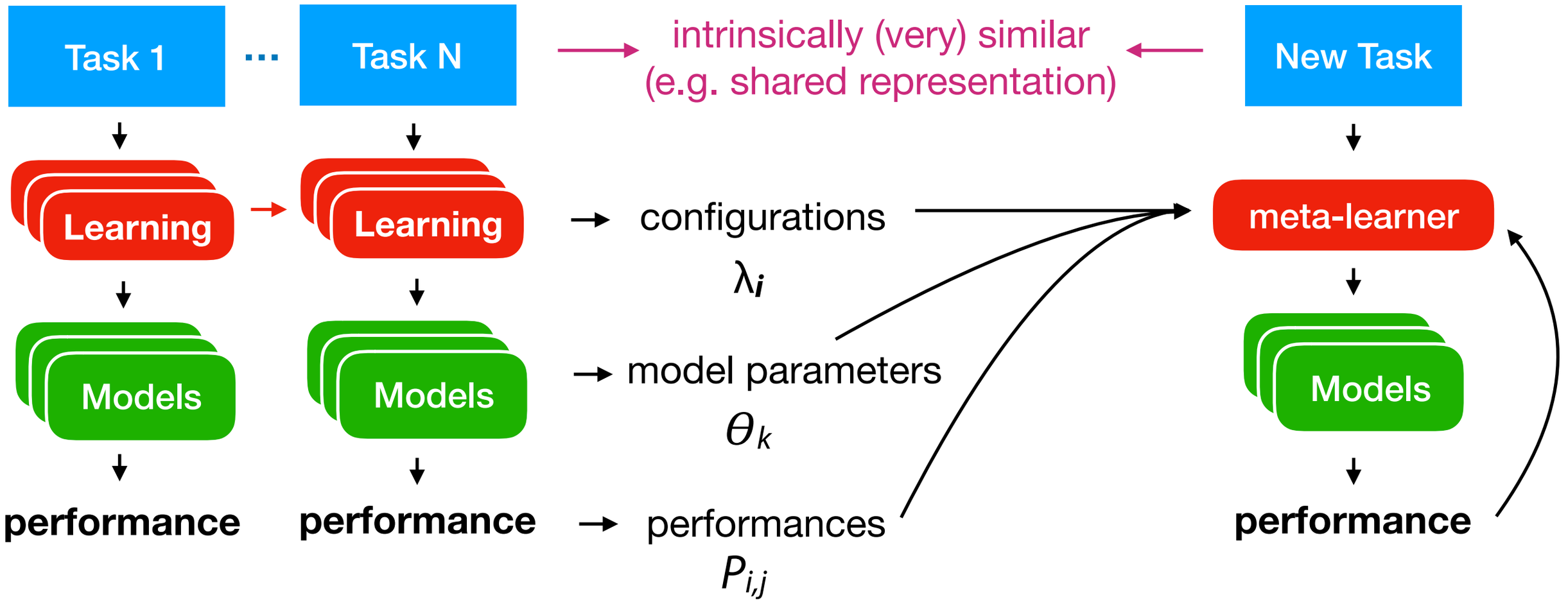
DARTS: Differentiable NAS

- Lots of further refinements
  - SNAS *[Xie et al 2019]*
    - Use Gumbel softmax (differentiable) on α_i
  - Proxyless NAS *[Cai et al 2019]*
    - Memory-efficient DARTS
      - trains sparse architectures only
  - …

Learning to learn from previous *models*

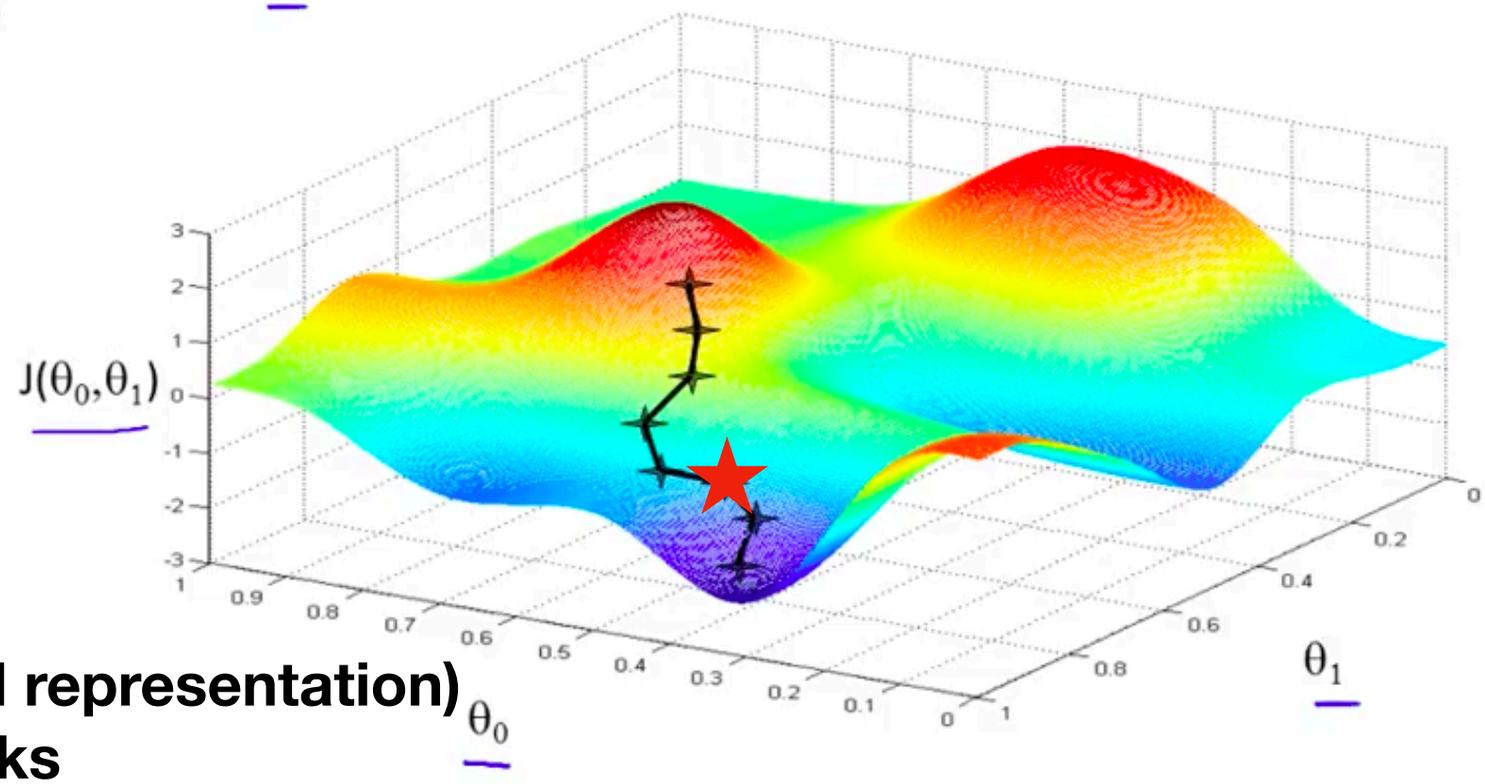
For models trained on intrinsically similar tasks

(model parameters, features,…)



Learning to learn

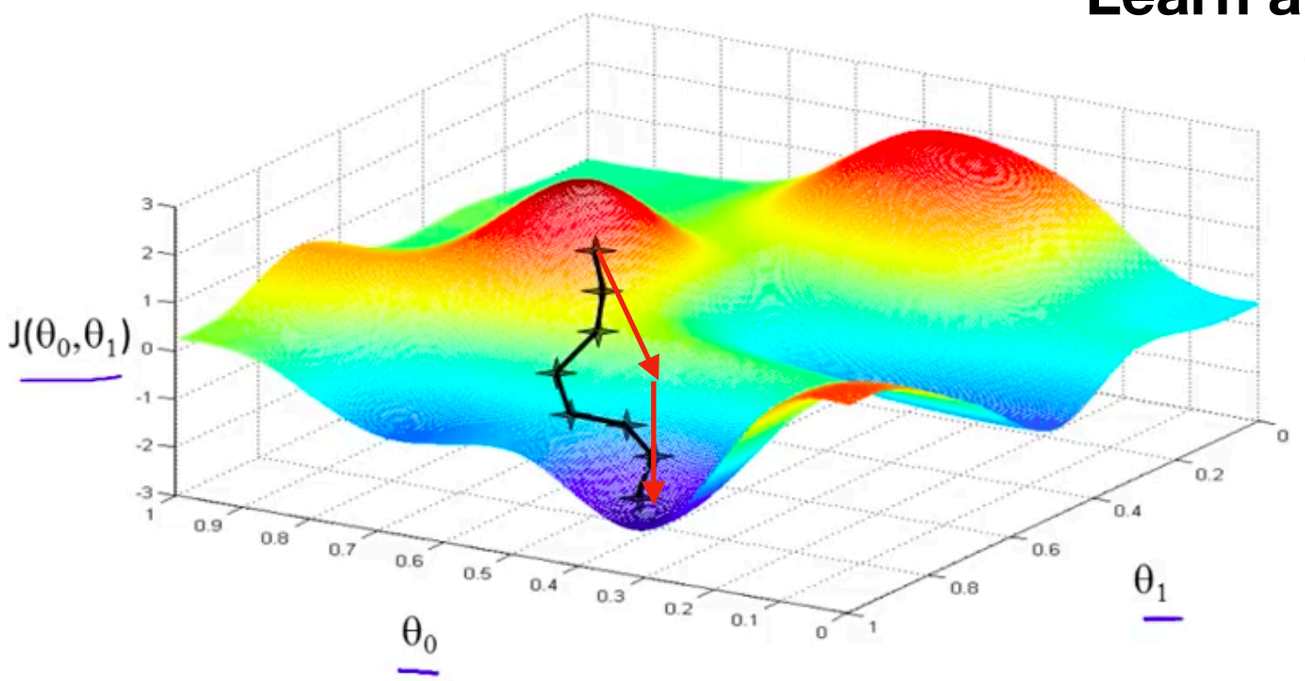
Learn a better gradient update rule for a group of tasks

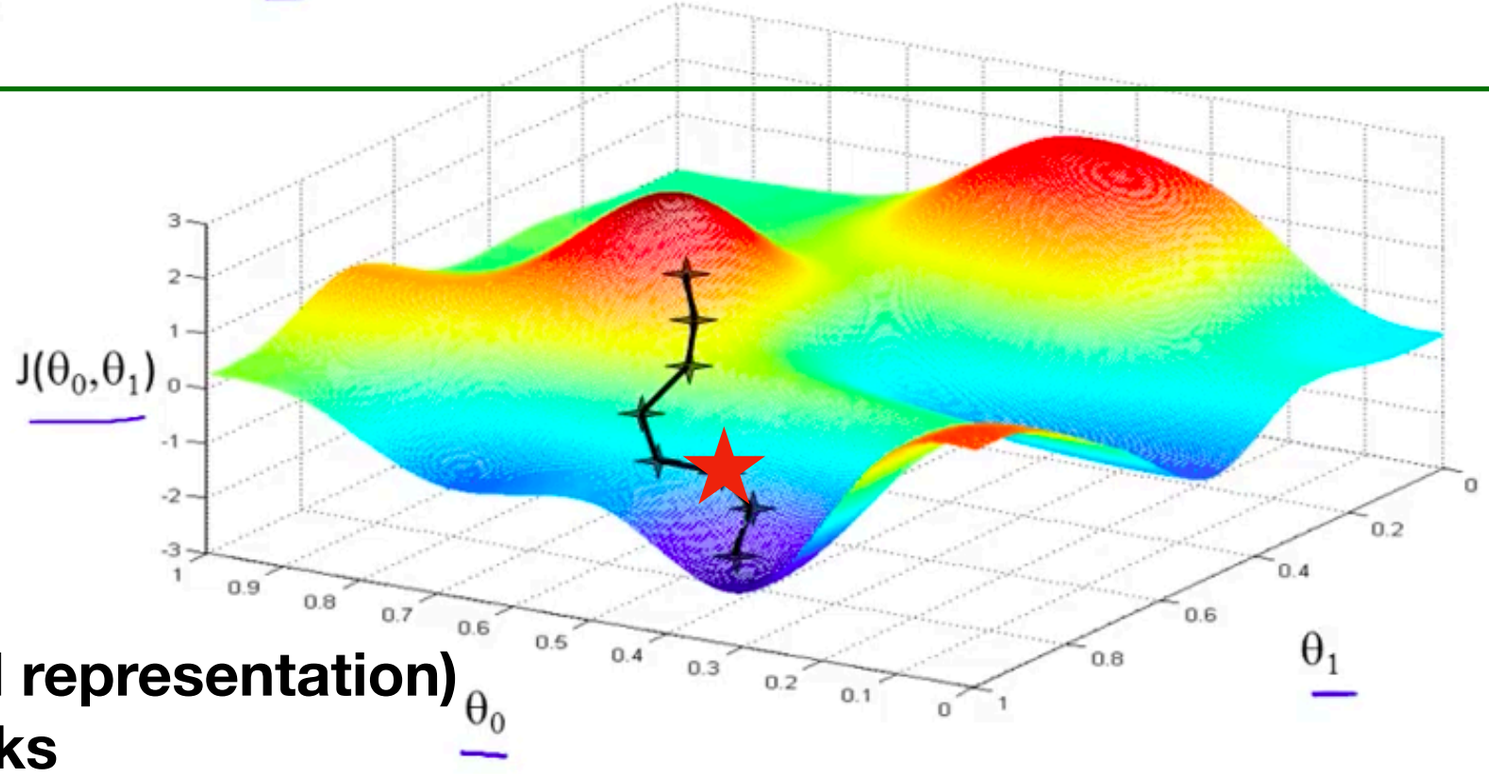


Learn a better initialization (and representation) for a group of tasks

Learning to learn

Learn a better gradient update rule for a group of tasks





Learn a better initialization (and representation) for a group of tasks

[1] Bengio et al. 1995

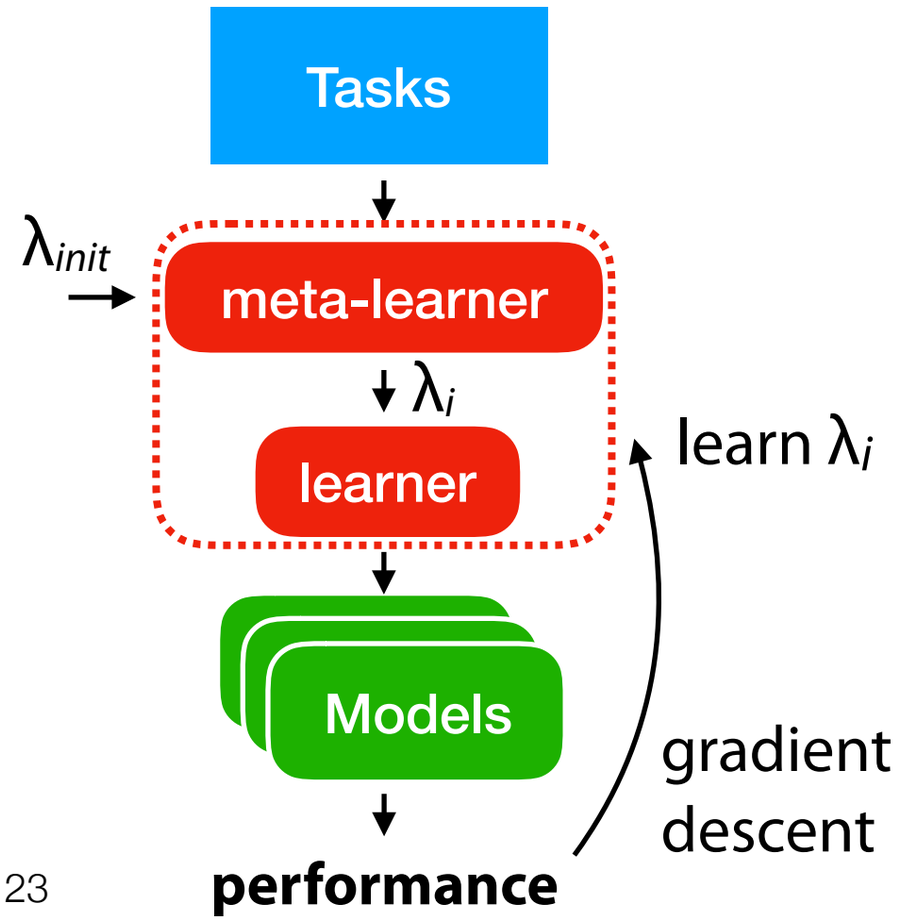
[2] Runarsson and Jonsson 2000

Learning to learn by gradient descent

- Our brains *probably* don't do backprop, replace it with:

$$\Delta w_{ij} = -\eta \frac{\partial E_p}{\partial w_{ij}}$$

  - Simple *parametric* (bio-inspired) rule to update weights [1]
  - Single-layer neural network to learn weight updates [2]
- Learn parameters across tasks, by gradient descent (meta-gradient)



Bengio et al.

$$\Delta \theta_i = \epsilon \, y_{pre(i)} \, k$$

presynaptic activity

learning rate

reinforcing signal

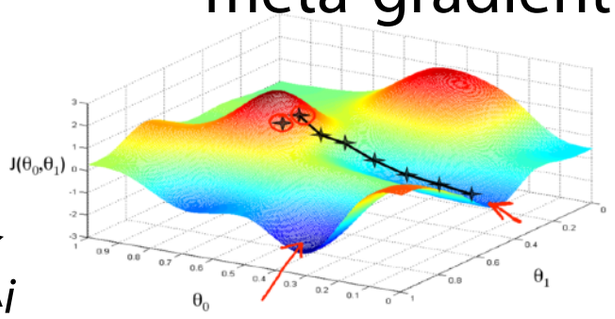
Runarsson and Jonsson

$$\Delta \theta_i = \eta \, (\quad)$$

learning rate

weights λ_i

meta-gradient

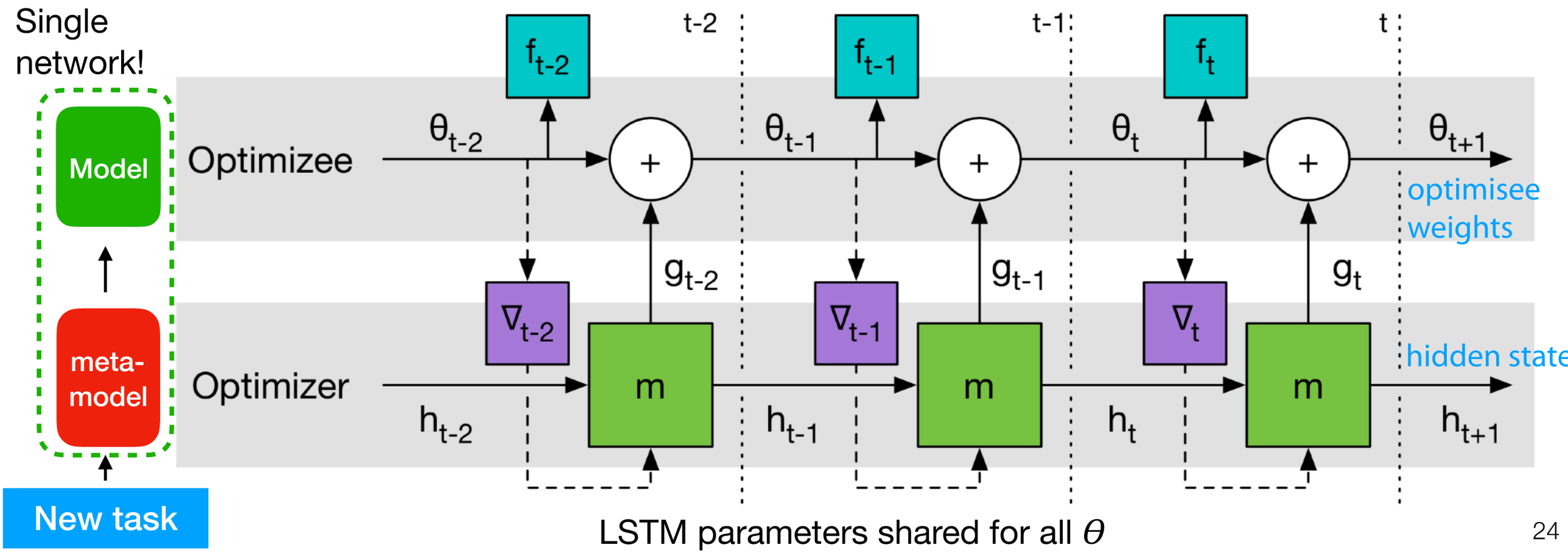


[1] Hochreiter 2001

[2] Andrychowicz et al. 2016

Learning to learn gradient descent by gradient descent

- Replace backprop with a recurrent neural net (LSTM)[1], not so scalable
- Use a coordinatewise LSTM [m] for scalability/flexibility (cfr. ADAM, RMSprop) [2]
  - Optimizee: receives weight update g_t from optimizer
  - Optimizer: receives gradient estimate ∇_t from optimizee
    - Learns how to do gradient descent across tasks



[1] Hochreiter 2001

[2] Andrychowicz et al. 2016

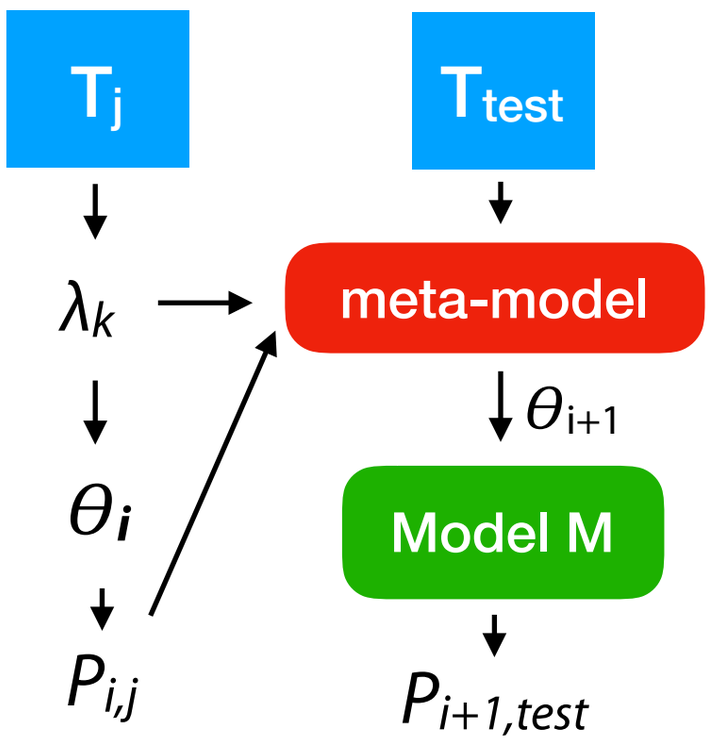
Learning to learn gradient descent by gradient descent

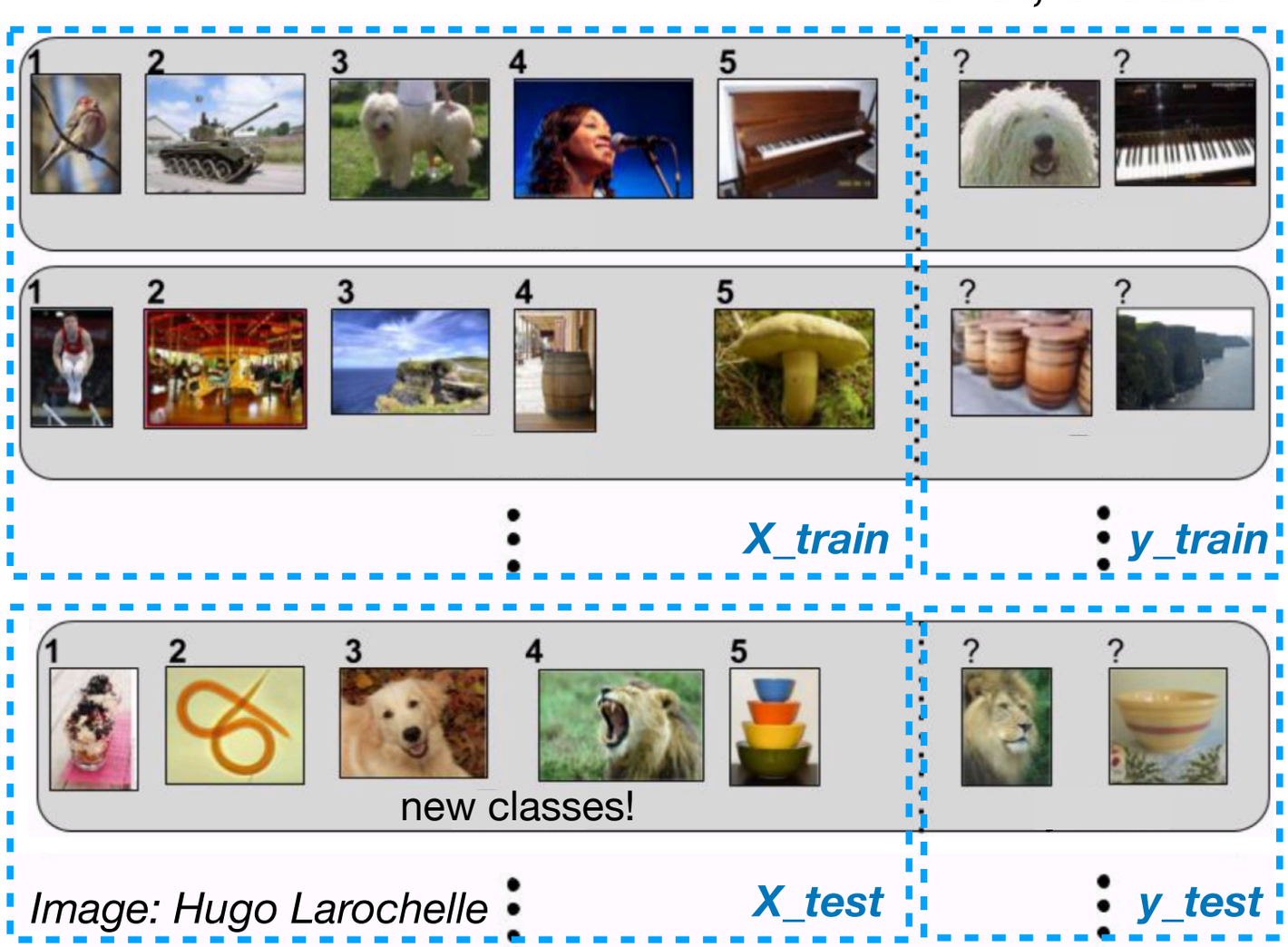
- Left: optimizer and optimizee trained to do style transfer
- Right: optimizer solves similar tasks (different style, content and resolution) without any more training data

Application: Few-shot learning

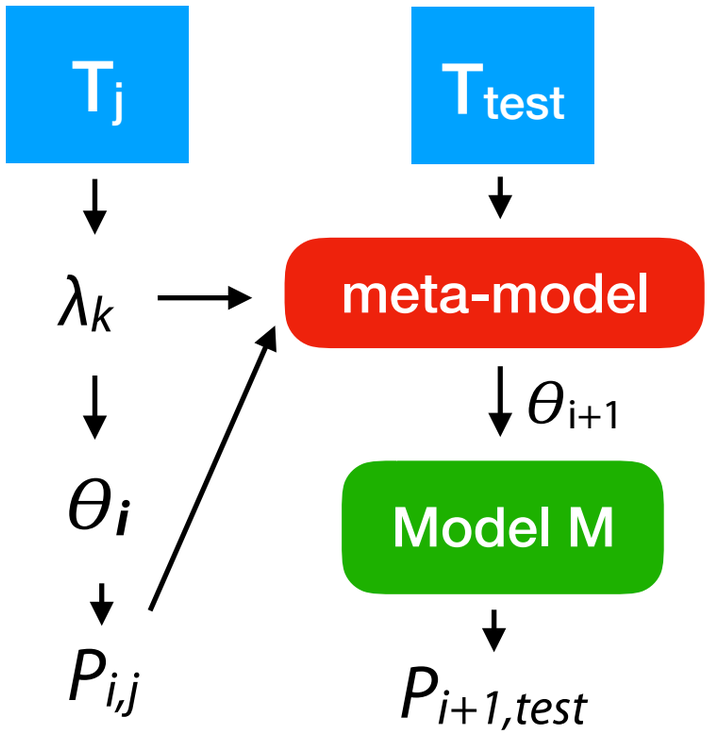
- Learn how to learn from few examples (given similar tasks)
  - Meta-learner must learn how to train a base-learner based on prior experience
    - Parameterize base-learner model and learn the parameters θ_i

$$Cost(\theta_i) = \frac{1}{|T_{test}|} \sum_{t \in T_{test}} loss(\theta_i, t)$$





Few-shot learning: approaches



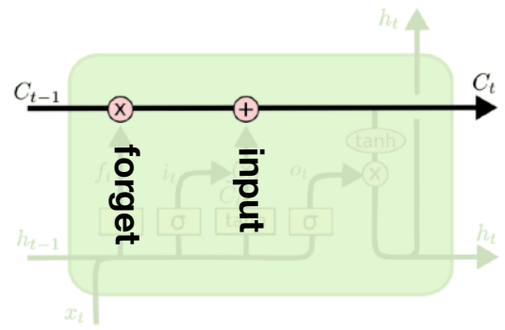
$$Cost(\theta_i) = \frac{1}{|T_{test}|} \sum_{t \in T_{test}} loss(\theta_i, t)$$

- Existing algorithm as meta-learner:
  - LSTM + gradient descent — *Ravi and Larochelle 2017*
  - Learn θ_{init} + gradient descent — *Finn et al. 2017*
  - kNN-like: Memory + similarity — *Vinyals et al. 2016*
  - Learn embedding + classifier — *Snell et al. 2017*
  - …
- Black-box meta-learner
  - Neural Turing machine (with memory) — *Santoro et al. 2016*
  - Neural attentive learner — *Mishra et al. 2018*
  - …

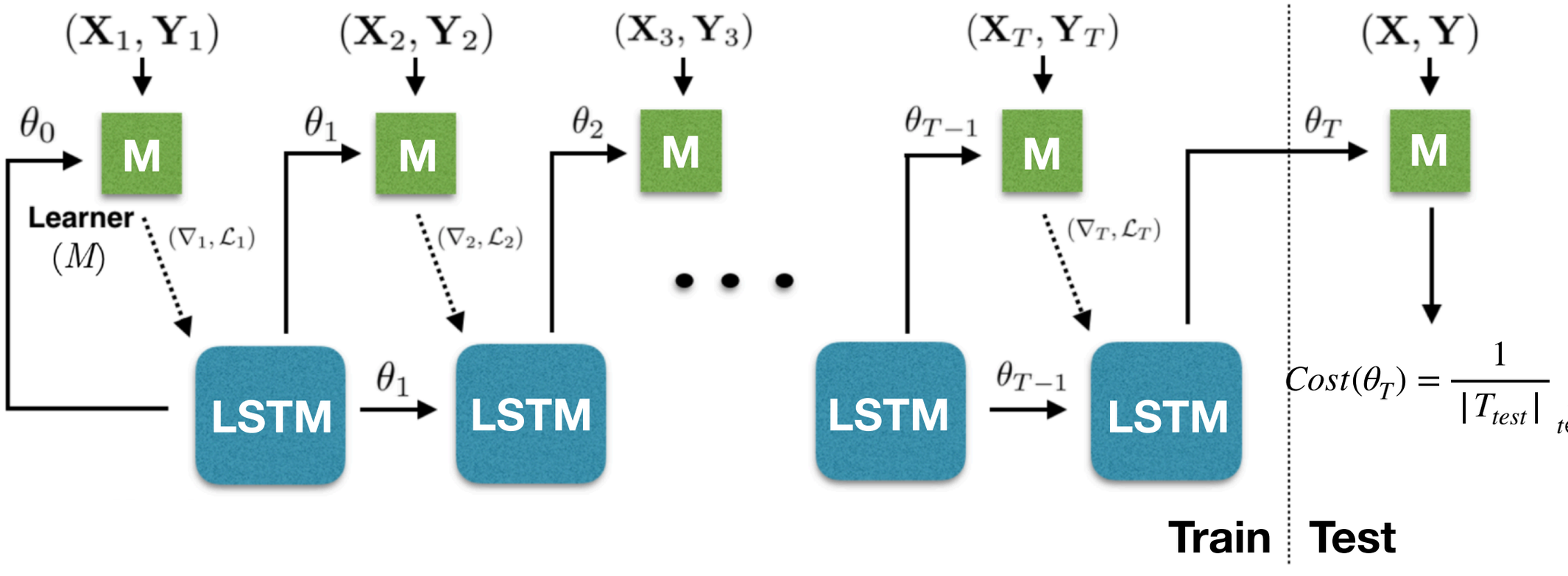
LSTM meta-learner + gradient descent

- Gradient descent update θ_t is similar to LSTM cell state update c_t

$$\theta_t = \theta_{t-1} - \alpha_t \nabla_{\theta_{t-1}} \mathcal{L}_t \qquad c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

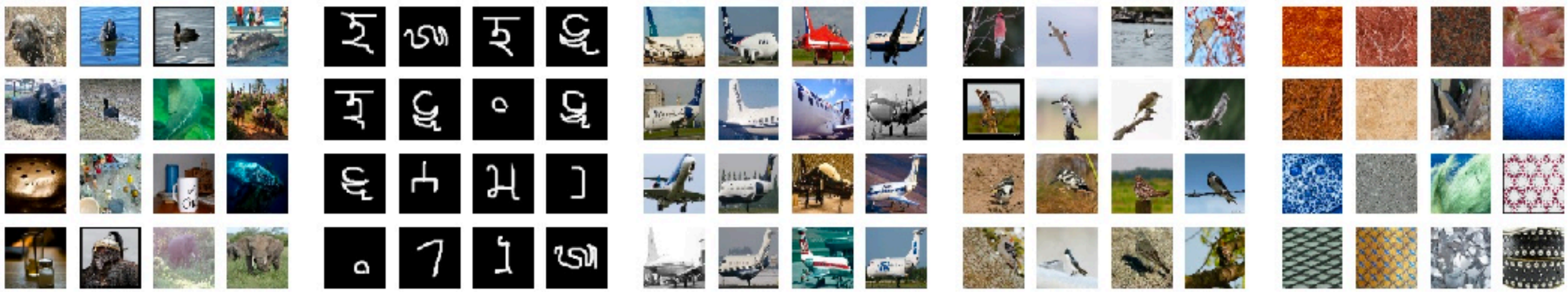


- Hence, training a meta-learner LSTM yields an update rule for training M
  - Start from initial θ_0, train model on first batch, get gradient and loss update
  - Predict θ_{t+1}, continue to t=T, get cost, backpropagate to learn LSTM weights, optimal θ_0

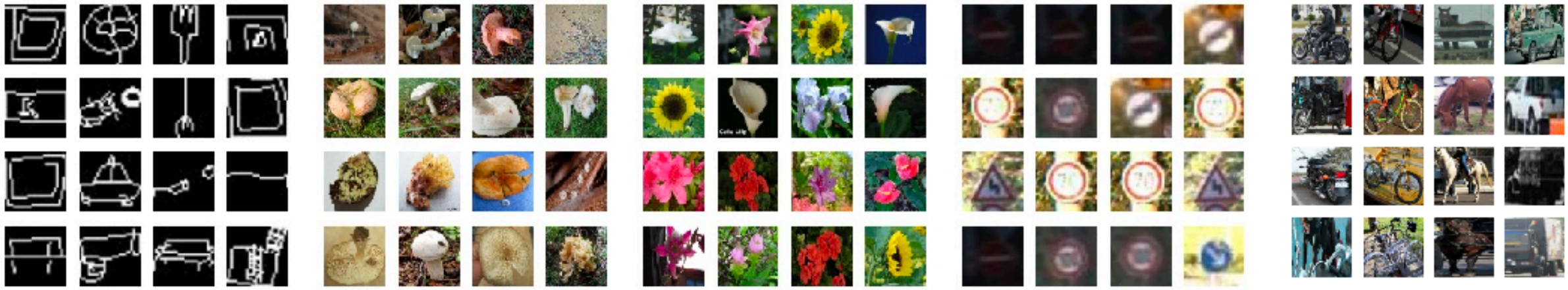


Triantafillou et al. 2019

Meta-dataset



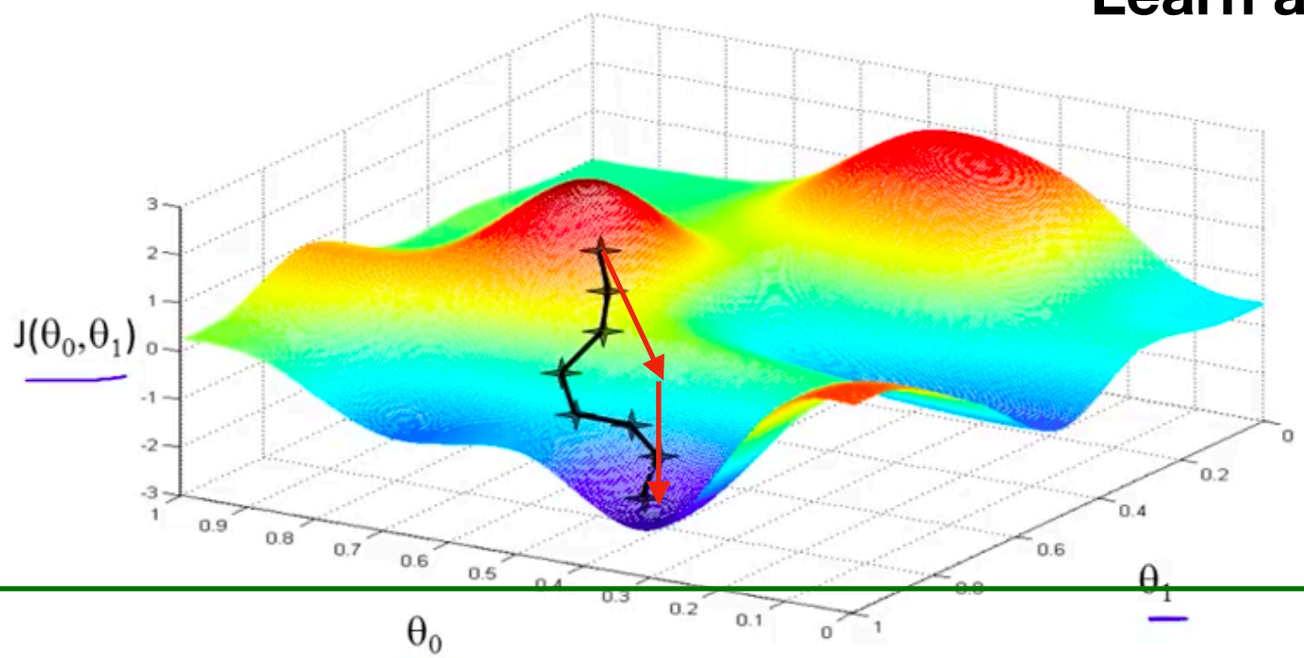
(a) ImageNet (b) Omniglot (c) Aircraft (d) Birds (e) DTD

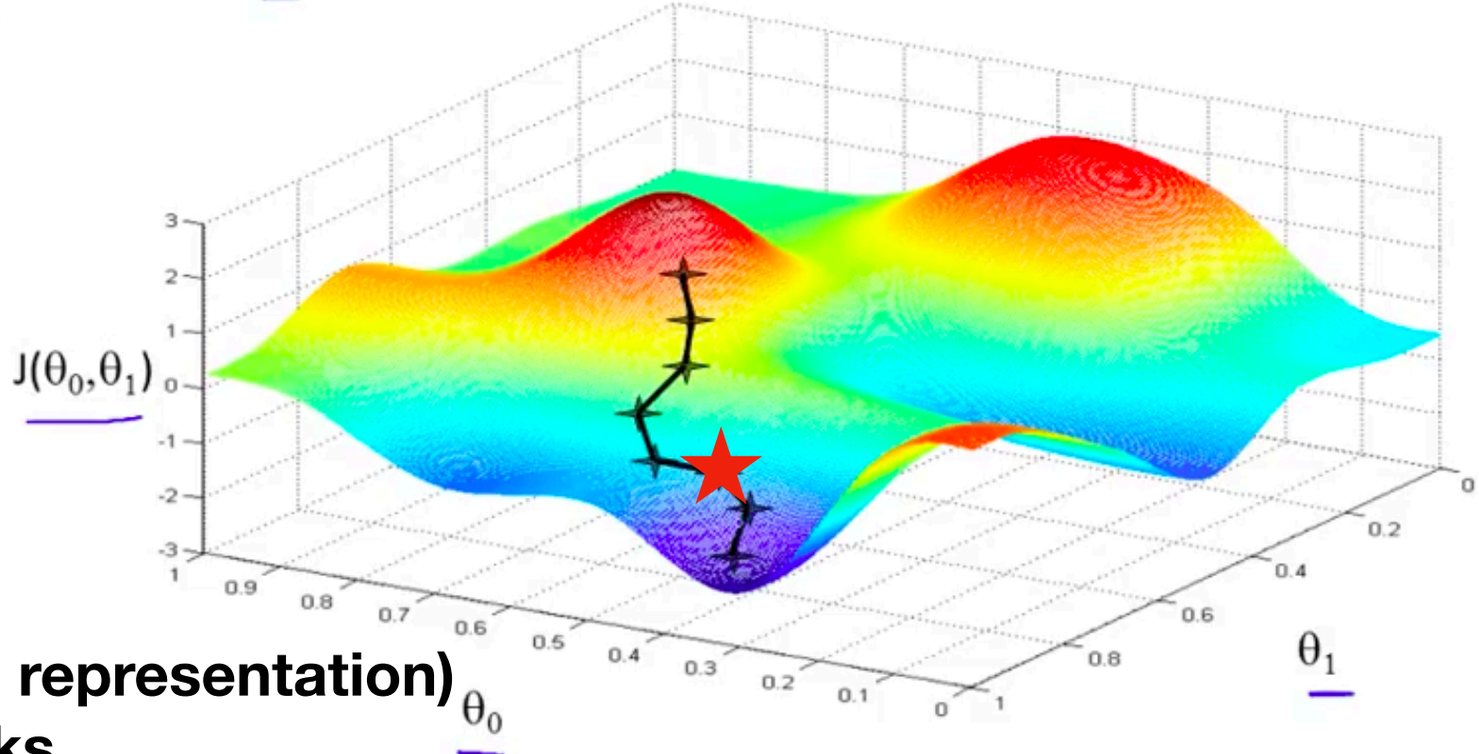


(f) Quick Draw (g) Fungi (h) VGG Flower (i) Traffic Signs (j) MSCOCO

Learning to learn

Learn a better gradient update rule for a group of tasks



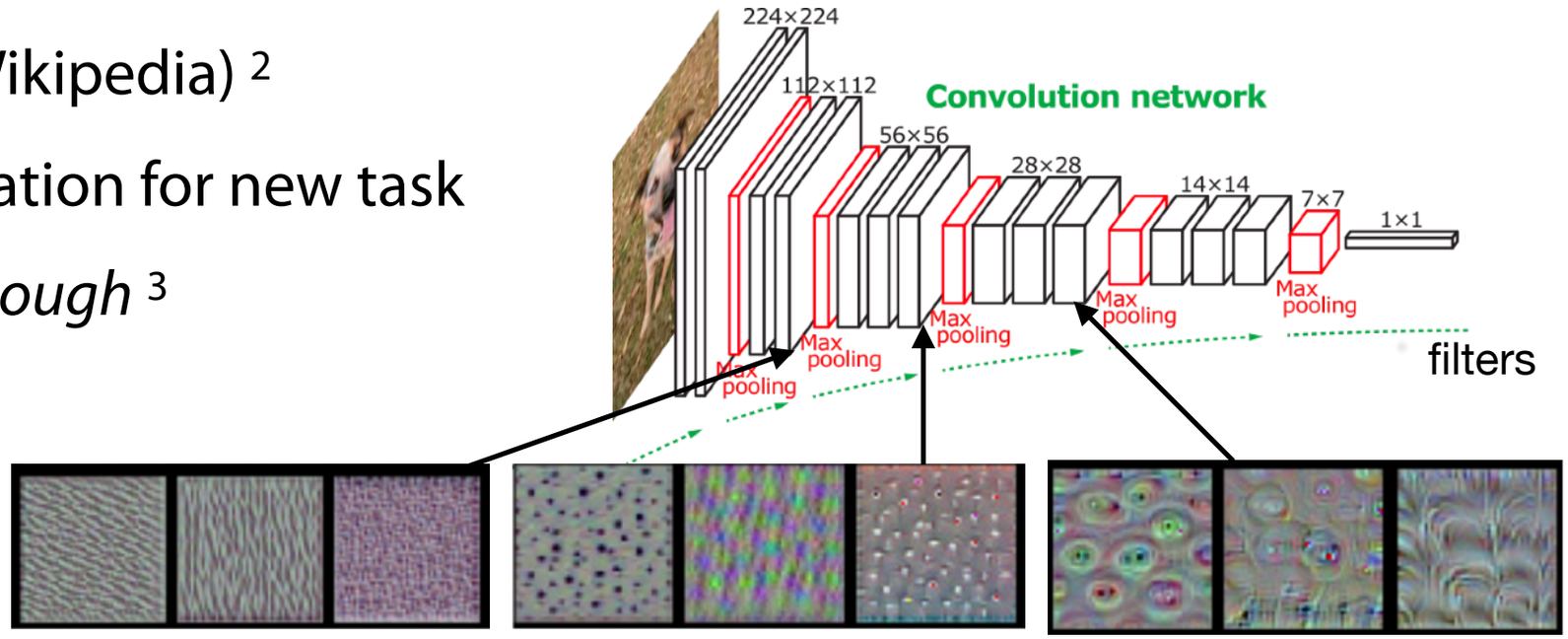


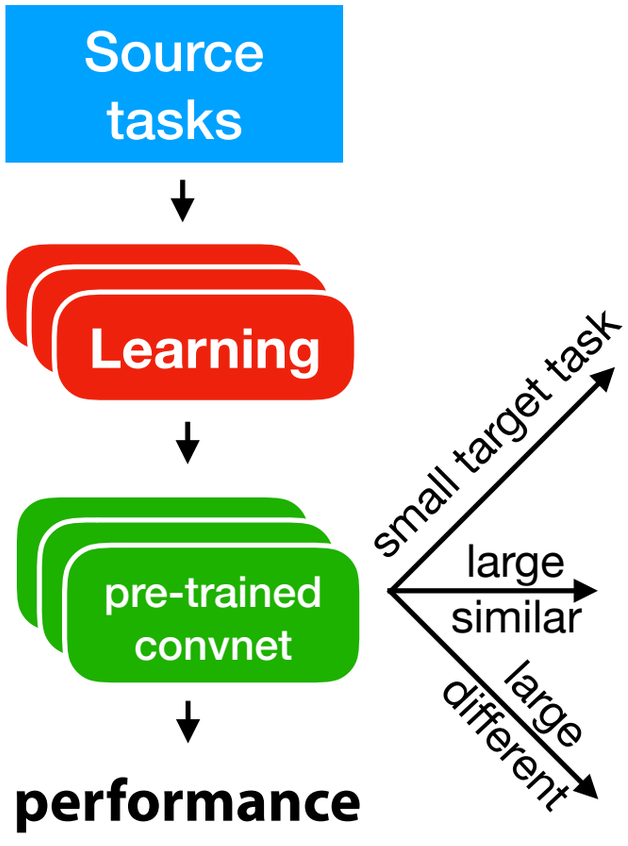
Learn a better initialization (and representation) for a group of tasks

[1] Razavian et al. 2014
[2] Mikolov et al. 2013
[3] Yosinski et al. 2014

Transfer learning

- Pre-train w*eights* from:
  - Large image datasets (e.g. ImageNet) [1]
  - Large text corpora (e.g. Wikipedia) [2]
- Use these weights as initialization for new task
- Fails if tasks are *not similar enough* [3]





Feature extraction:
remove last layers, use output as features
if task is quite different, remove more layers

End-to-end tuning:
train from initialized weights

Fine-tuning:
unfreeze last layers, tune on new task

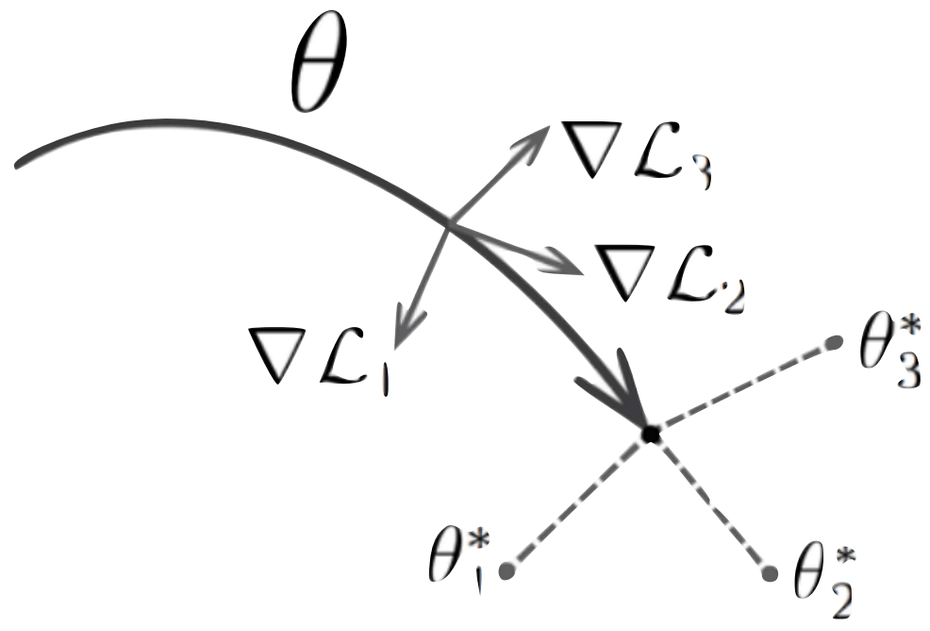
[1] Finn et al. 2017

Model-agnostic meta-learning

- Solve new tasks faster by learning a model *initialization* from similar tasks
  - Current initialization θ
  - On K examples/task, evaluate $\nabla_\theta L_{T_i}(f_\theta)$
  - Update weights for $\theta_1, \theta_2, \theta_3$
  - Update θ to minimize sum of per-task losses

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}\left(f_{\theta_i'}\right)$$

  - Repeat



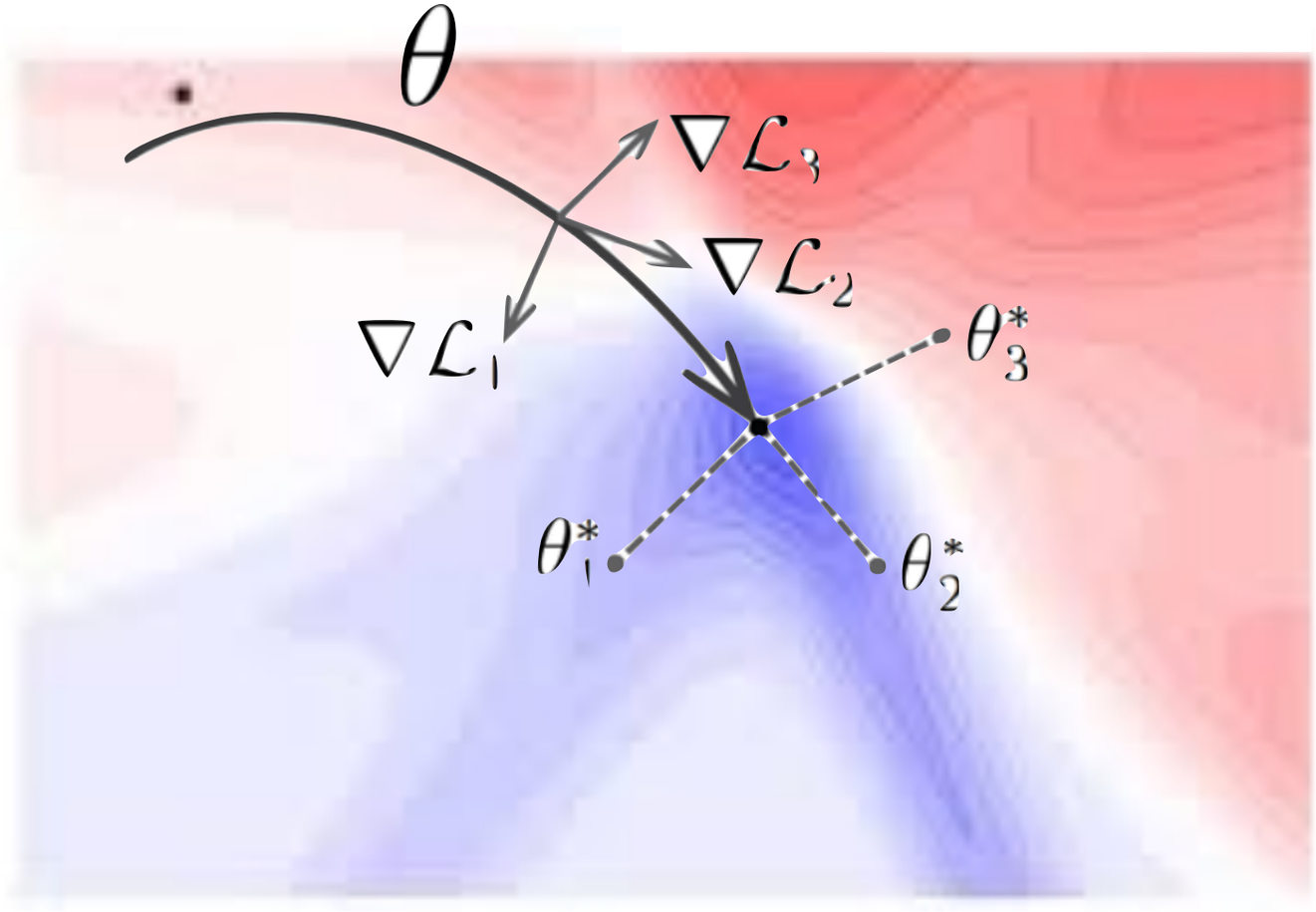
[1] Finn et al. 2017

Model-agnostic meta-learning

- Solve new tasks faster by learning a model *initialization* from similar tasks
  - Current initialization θ
  - On K examples/task, evaluate $\nabla_\theta L_{T_i}(f_\theta)$
  - Update weights for $\theta_1, \theta_2, \theta_3$
  - Update θ to minimize sum of per-task losses

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}\left(f_{\theta_i'}\right)$$

  - Repeat



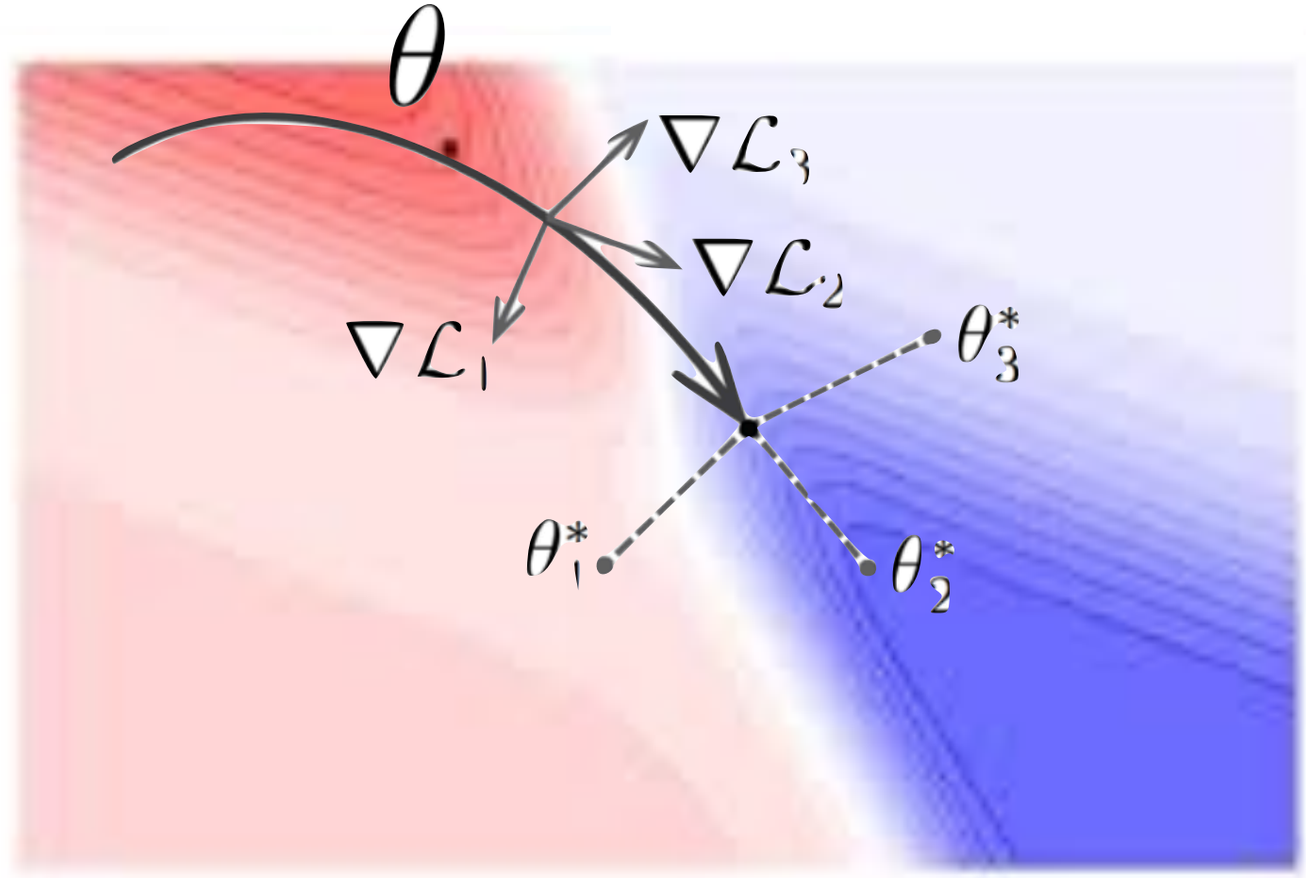
[1] Finn et al. 2017

Model-agnostic meta-learning

- Solve new tasks faster by learning a model *initialization* from similar tasks
  - Current initialization θ
  - On K examples/task, evaluate $\nabla_\theta L_{T_i}(f_\theta)$
  - Update weights for $\theta_1, \theta_2, \theta_3$
  - Update θ to minimize sum of per-task losses

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}\left(f_{\theta_i'}\right)$$

  - Repeat



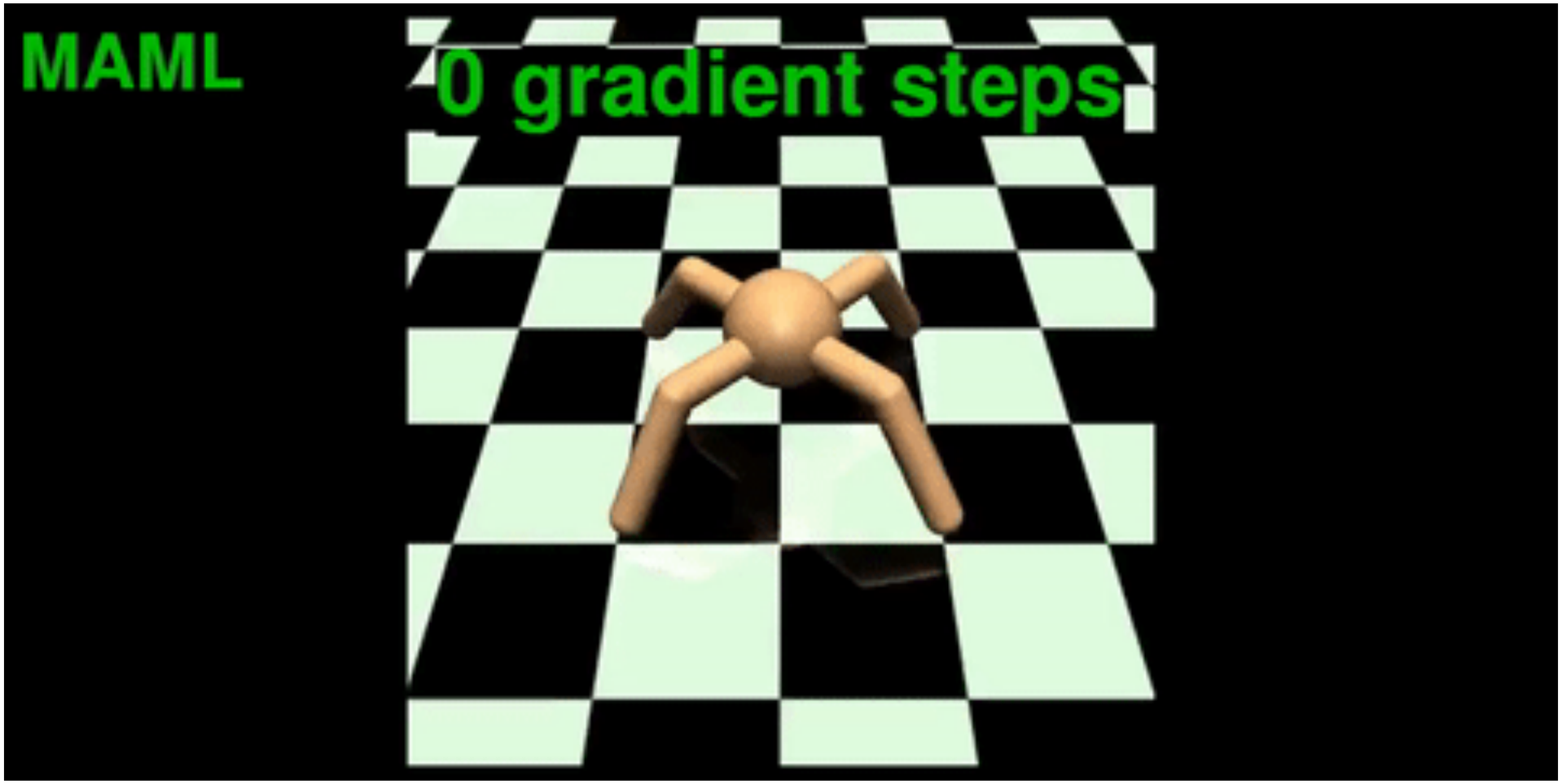
[1] Finn et al. 2018
[2] Nichol et al. 2018

Model-agnostic meta-learning

- More resilient to overfitting
- Generalizes better than LSTM approaches for few shot learning
- *Universality* [1]: no theoretical downsides in terms of expressivity when compared to alternative meta-learning models.
- Many variants and applications: *Finn & Levine 2019*
  - REPTILE: do SGD for k steps in one task, only then update init. weights [2]
  - PLATIPUS: probabilistic MAML
  - Bayesian MAML (Bayesian ensemble)
  - Online MAML
  - …

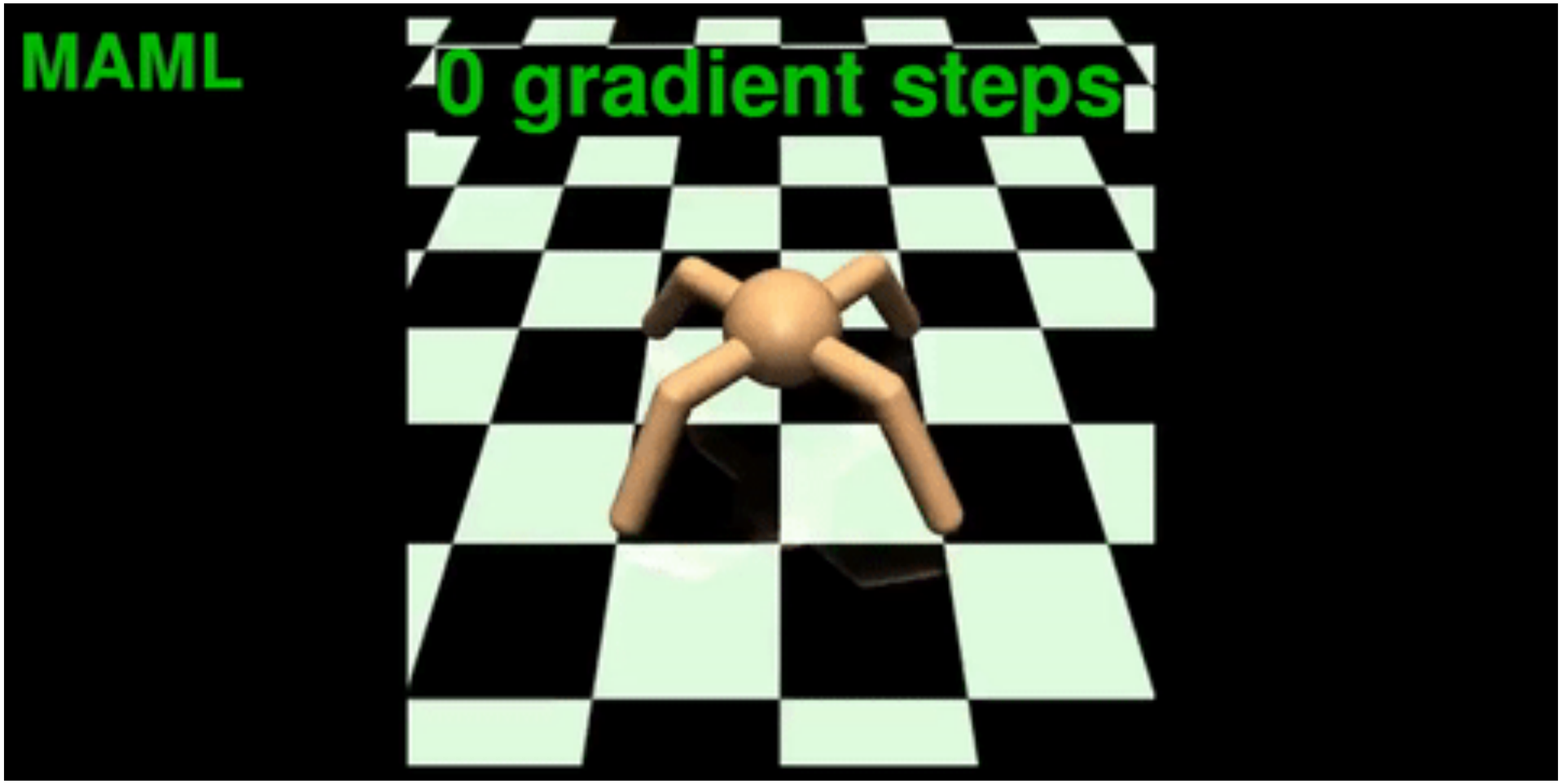
Model-agnostic meta-learning

- For reinforcement learning:



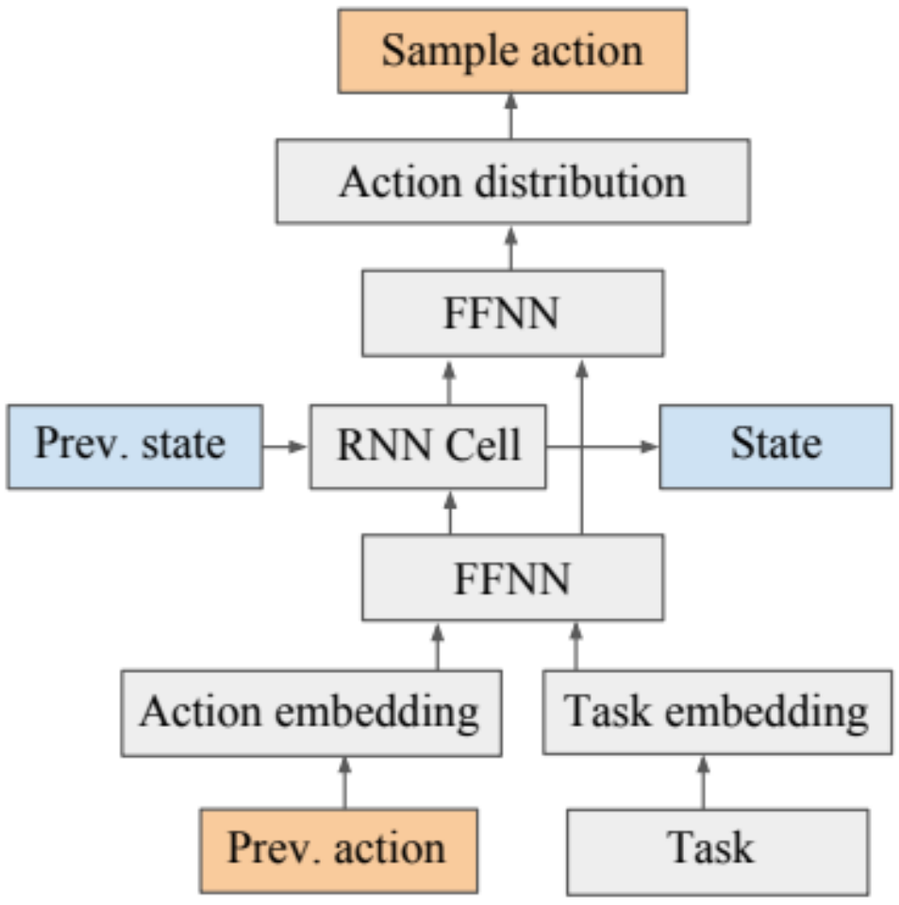
Model-agnostic meta-learning

- For reinforcement learning:



Neural Architecture Meta-Learning

- Warm-start a deep RL controller based on prior tasks *Wong et al. 2018*
- Much faster than single-task equivalent

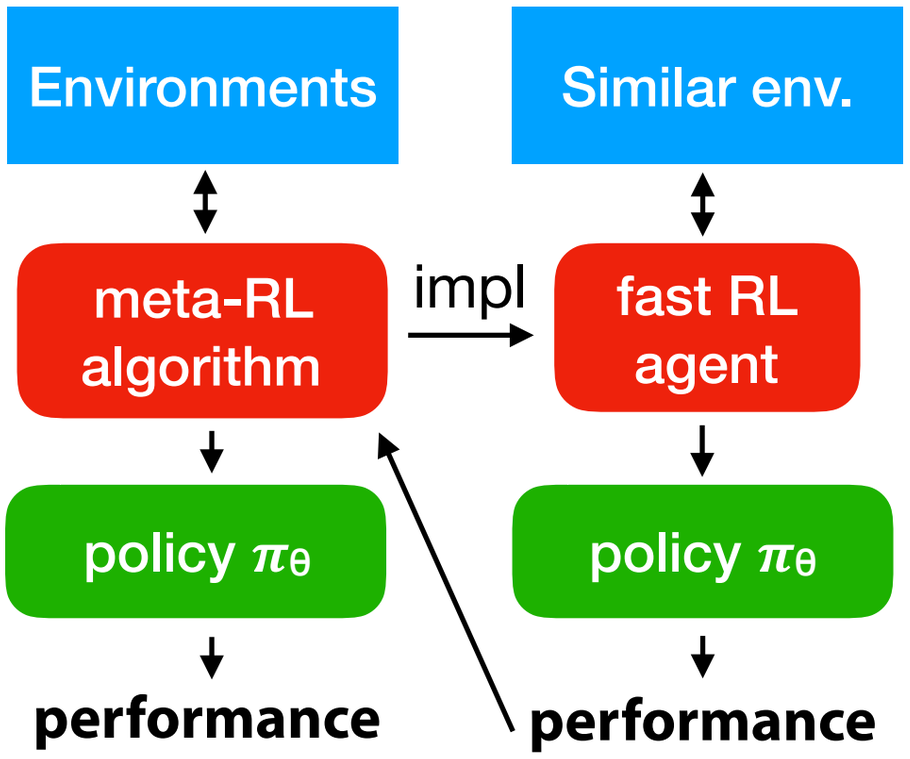


- Another idea: warm-start DARTS with task-specific one-shot model

[1] Duan et al. 2017
[2] Wang et al. 2017
[3] Duan et al. 2017

Learning to reinforcement learn

- Humans often learn to play new games much faster than RL techniques do
- Reinforcement learning is very suited for learning-to-learn:
  - Build a learner, then use performance as that learner as a reward

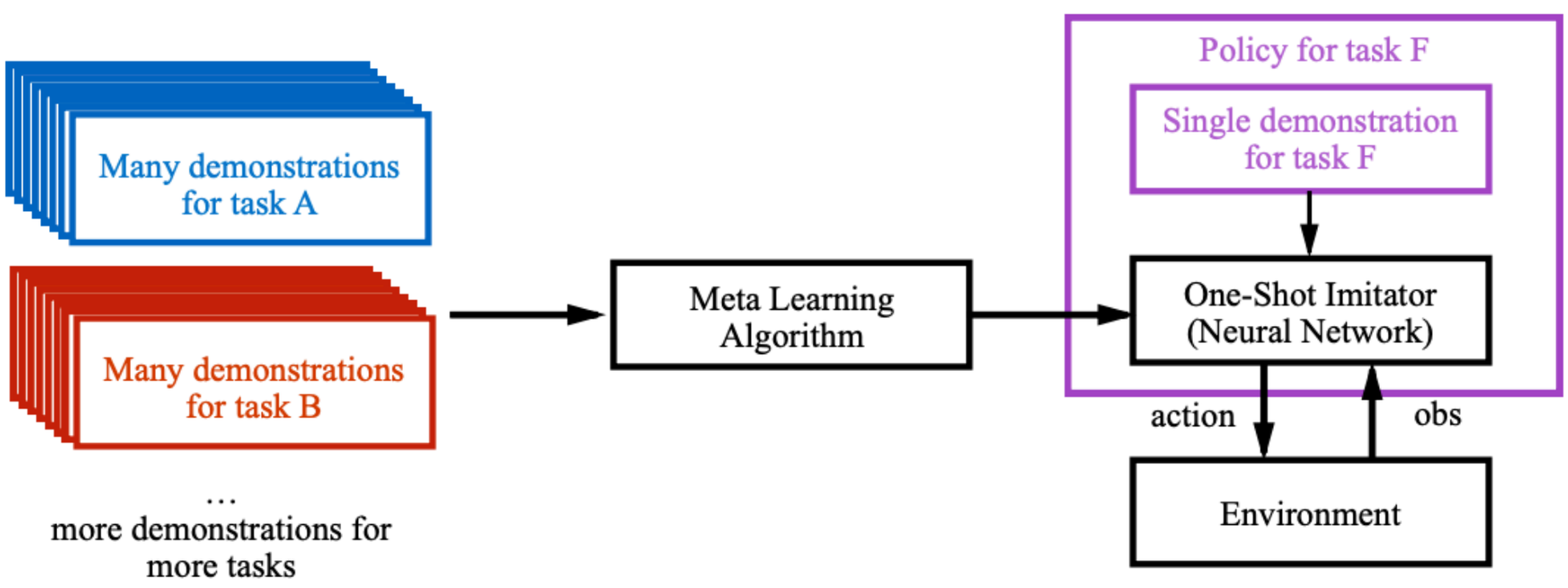


- Learning to reinforcement learn [1,2]
  - Use RNN-based deep RL to train a recurrent network on many tasks
  - Learns to implement a ‘fast’ RL agent, encoded in its weights

[1] Duan et al. 2017
[2] Wang et al. 2017
[3] Duan et al. 2017

Learning to reinforcement learn

- Also works for few-shot learning [3]
  - Condition on observation + upcoming demonstration
- You don't know what someone is trying to teach you, but you prepare for the lesson



Learning to learn more tasks

- Active learning — *Pang et al. 2018*
  - Deep network (learns representation) + policy network
  - Receives state and reward, says which points to query next
- Density estimation — *Reed et al. 2017*
  - Learn distribution over small set of images, can generate new ones
  - Uses a MAML-based few-shot learner
- Matrix factorization — *Vartak et al. 2017*
  - Deep learning architecture that makes recommendations
  - Meta-learner learns how to adjust biases for each user (task)
- Replace hand-crafted algorithms by learned ones.
- Look at problems through a meta-learning lens!

Meta-data sharing building a shared memory

- What if we have a shared memory of all machine learning experiments?
- OpenML.org
  - Thousands of uniform datasets, 100+ meta-features
  - Millions of evaluated runs
    - Same splits, 30+ metrics
    - Traces, models *(opt)*
- **APIs in Python, R, Java,… Sklearn, Keras, PyTorch, MXNet,…**
- Publish your own runs
- **Never ending learning**
- Benchmarks

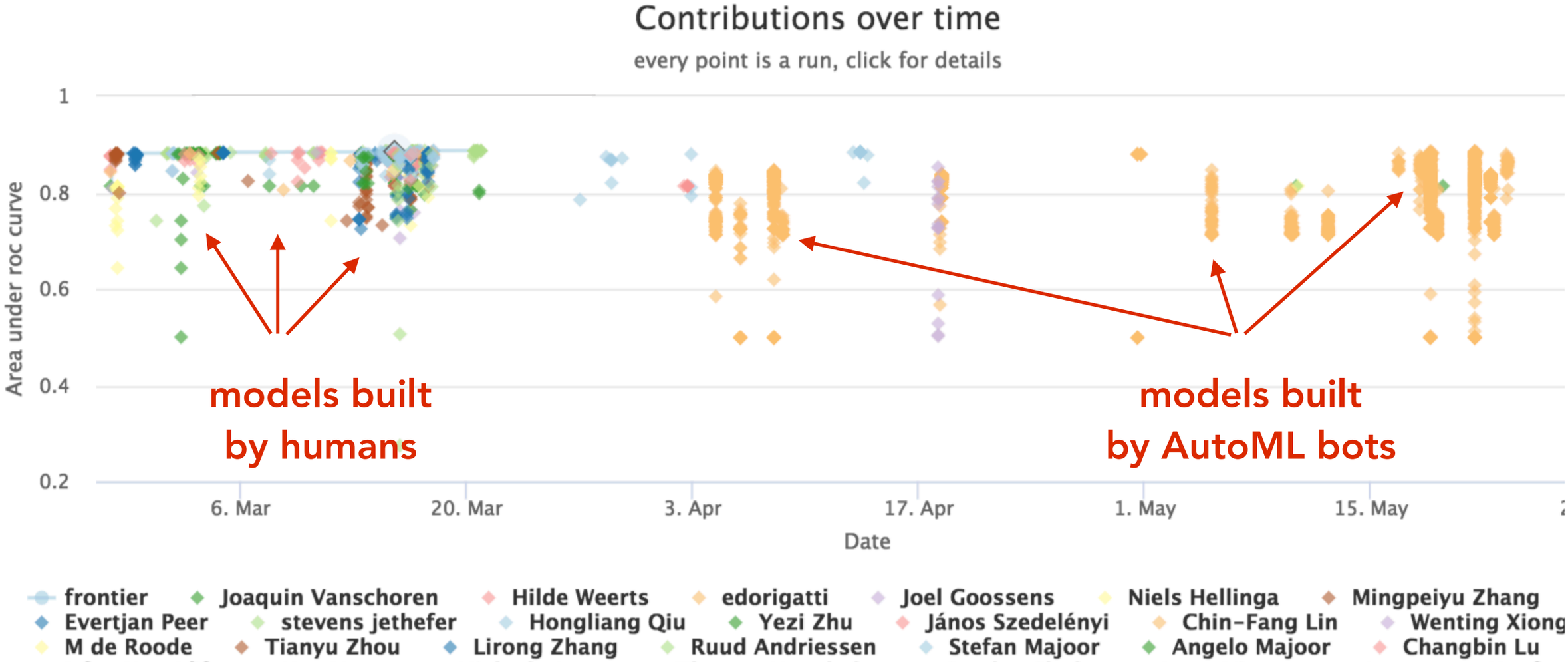
```
import openml as oml
from sklearn import tree

task = oml.tasks.get_task(14951)
clf = tree.ExtraTreeClassifier()
flow = oml.flows.sklearn_to_flow(clf)
run = oml.runs.run_flow_on_task(task, flow)
myrun = run.publish()
```

run *locally*, share *globally*

Meta-data sharing building a shared memory

- Train and run AutoML bots on any OpenML dataset
- Never-ending learning (from robots and humans)

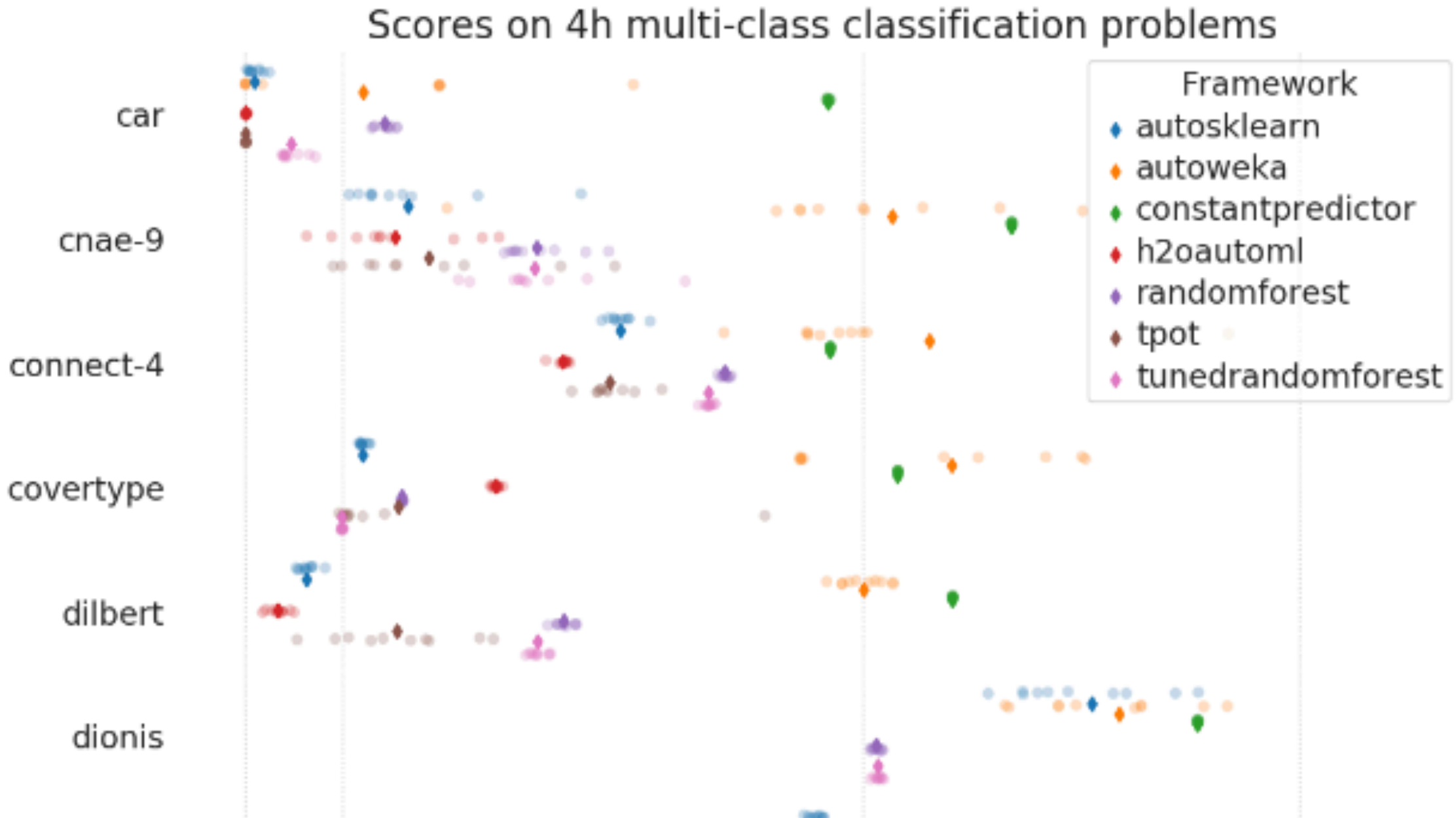


Gijsbers et al. 2019

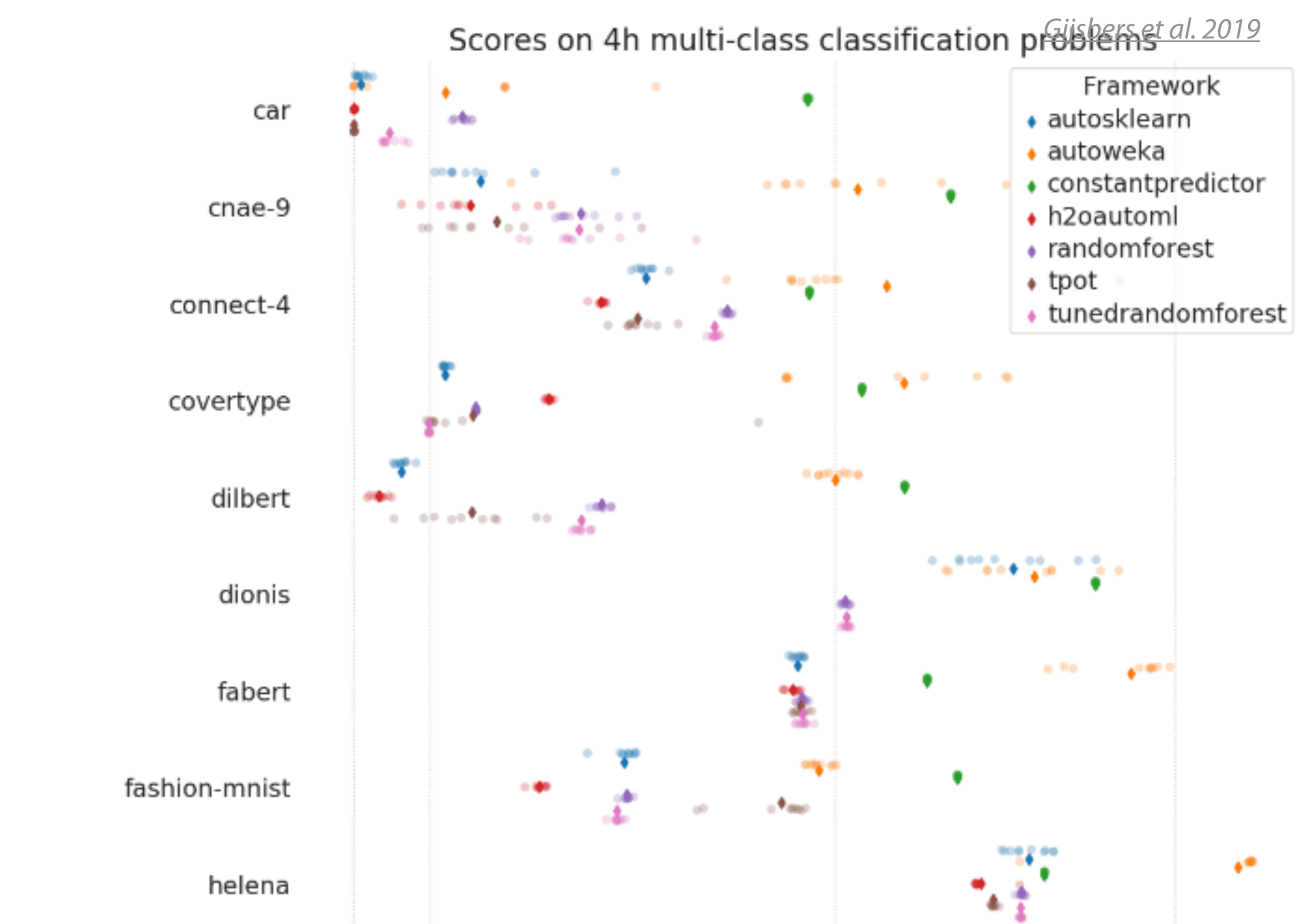
AutoML tools and benchmarks

Open source benchmark on https://openml.github.io/automlbenchmark/

- On OpenML datasets, will regularly be updated, anyone can add



Gijsbers et al. 2019



AutoML open source tools

	Architecture search	Hyperparameter search	Improvements	Meta-learning
Auto-WEKA	Parameterized pipeline	Bayesian Optimization (RF)		
auto-sklearn	Parameterized pipeline	Bayesian Optimization (RF)	Ensembling	warm-start
BO-HB	Parameterized pipeline	Tree of Parzen Estimators	Ensembling, Hyperband	
hyperopt-sklearn	Parameterized pipeline	Tree of Parzen Estimators		
TPOT	Evolving pipelines (trees)	Genetic programming		
GAMA	Evolving pipelines (trees)	Asynchronous evolution	Ensembling, Hyperband	
H2O AutoML	Single algorithms	random search	Stacking	
ML-Plan	Planning-based pipeline	Hierarchical planner		
OBOE	Single algorithms	Low rank approximation	Ensembling	Runtime predictor

AutoML tools and benchmarks

Observations so far (for pipeline search):

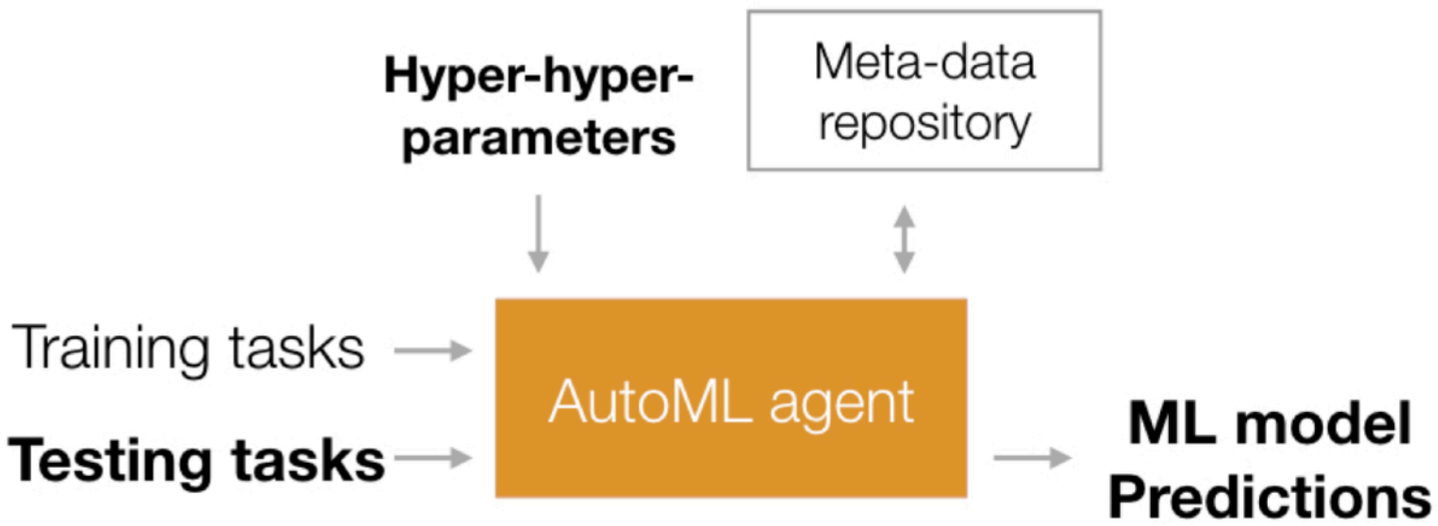
- No AutoML system outperforms all others (Auto-WEKA is worse)
- Auto-sklearn, H2O, TPOT can outperform each other on given datasets
- Tuned RandomForests are a strong baseline
- High numbers of classes, unbalanced classes, are a problem for most tools
- Little support for anytime prediction …

We need good benchmarks for NAS as well…

AutoML Gym

Environment to train general-purpose AutoML algorithms

- Open benchmark, meta-learning, continual learning, efficient NAS



Open PostDoc position!
http://bit.ly/automl-gym

Towards human-like learning to learn

- Learning-to-learn gives humans a significant advantage
  - **Learning how to learn any task empowers us far beyond knowing how to learn specific tasks.**
  - It is a **universal** aspect of life, and how it evolves
- Very exciting field with many unexplored possibilities
  - Many aspects not understood (e.g. task similarity), need more experiments.
- **Challenge**:
  - Build learners that **never stop learning**, that **learn from each other**
  - *Fast* learning by *slow* meta-learning
  - Build a ***global memory*** for learning systems to learn from
  - **Let them explore / experiment by themselves**

Thank you!

Never stop learning

more to learn

http://www.automl.org/book/

Chapter 2: Meta-Learning

special thanks to

Pavel Brazdil, Matthias Feurer, Frank Hutter, Erin Grant, Hugo Larochelle, Raghu Rajan, Jan van Rijn, Jane Wang